EFFECTIVE FAULT LOCALIZATION TECHNIQUES FOR CONCURRENT SOFTWARE

A Thesis
Presented to
The Academic Faculty

by

Sang Min Park

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2014

EFFECTIVE FAULT LOCALIZATION TECHNIQUES FOR CONCURRENT SOFTWARE

Approved by:

Dr. Richard Vuduc,
Committee Chair and Advisor
College of Computing
Georgia Institute of Technology

Dr. Alessandro Orso
College of Computing
Georgia Institute of Technology

Dr. Mary Jean Harrold, Advisor
College of Computing
Georgia Institute of Technology

Dr. Milos Prvulovic
College of Computing
Georgia Institute of Technology

Dr. Mayur Naik
College of Computing
Georgia Institute of Technology

Dr. Mark Grechanik
Department of Computer Science
University of Illinois at Chicago

Date Approved: 6 August 2014

ACKNOWLEDGEMENTS

First of all, I must express my utmost gratitude to my advisor, Dr. Mary Jean Harrold, for her unlimited support. Mary Jean helped me in all aspects of research from finding a research topic to writing a good research paper. Also, she gave me advice to be a good researcher as well as a good person.

I am deeply indebted to Dr. Richard Vuduc. Rich has been a good research mentor from the early stage of my Ph.D. After Mary Jean's passing, he graciously took me as his student and guided me to finish the Ph.D. journey.

I also thank other committee members, Drs. Mark Grechanik, Alex Orso, Mayur Naik, and Milos Prvulovic, for their useful comments to complete this research. Especially, I thank Mark for giving me an opportunity to work at a research lab and for showing me his outlook to research.

During my internships at Accenture Technology Labs, KAIST, and NEC Labs, I was fortunate to work with wonderful researchers, Drs. Chen Fu, Qing Xie, Christoph Csallner, Moonzoo Kim, Malay Ganai, and Aarti Gupta. I was enlightened by their view of real software engineering problems in industry. I thank Shin Hong, Gigon Bae, and Yunho Kim for interesting discussions at KAIST.

I am very grateful to my undergraduate advisors, Drs. Doo-Hwan Bae and Yong-Rae Kown, at KAIST. Dr. Kwon introduced me to Mary Jean and motivated me to pursue a Ph.D. at Georgia Tech. Dr. Bae gave me the opportunity to work in his research lab and learn practical aspects of Software Engineering. His invaluable advice has helped me since even when I struggled during my Ph.D.

I thank my friends in the SE group: Shauvik Roy Choudhary and Wei Jin, my best friends, for sharing all good and bad memories; Chaitanya Parakash Namburi

for being my English teacher during the first two years; Mijung Kim for insightful research discussions in Korean; Hina Shah for being a kind officemate; Juyuan Yang for the help in preparation for interviews; Chris Parnin for mentoring my user study; and others, Saswat Anand, Aliva Pattnaik, George Baah, Raul Santelices, Paul Li, Jake Cobb, Mattia Fazzini, Jim Jones, Xiangyu Li, Qianqian Wang, and Jie Lu for their companionship. I thank friends in the HPG Garage group, Cong Hou, Jeehwan Choi, and Kent Czechowski. I will always remember precious memories with you.

I had fun with my Korean friends at GT. Thanks Minjang Kim for productive research discussions; Minsung Jang for discussing all issues in life and research; Myunghyun Ha for being my roommate for two years; Myungcheol Doo for being my roommate during my Accenture internship; Younggyun Koh, my landlord, for supporting me to finish my PhD; and other friends, Sunjae Park, Jungju Oh, Hyojoon Kim, Seungyeon Kim, Jieun Lim, Joonseok Lee, Soo Kyung Kim, Changhyun Choi, Ilho Song, Moonkyung Ryu, Jaegul Choo, Kirak Hong, Stacy Kim, Ja-Young Sung, Unkyong Lee, Sean Hay Kim, Hae Youn Joung, Chankyu Han, Ho Young Kim, and Kyungho Jeon. I also thank my high school alumni, Seunghee Kim, Jung Hyun Hong, Yusun Lim, Jiyo Shin, Sang Gyun Park, and Wonhee Cho, for occasional reunions.

I am fortunate to have lifelong friends in Korea. My high school friends, Ilhwan Song, Hyungjune Lim, Heebong Park, Sang Mo Park, Donghyun Kim, Ildoo Kim, Wonjae Lee, and Wonjoon Jo, always welcomed me whenever I visited Korea. The members of the CoC KMGNGD clan, especially the leader, Min Gu Kang, have become good comrades in battles against anonymous enemies.

Lastly, I express my sincere gratitude to my parents. This dissertation would not have been successfully completed without their patience. I always love you.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . . **iii**

LIST OF TABLES . . . . . **ix**

LIST OF FIGURES . . . . . **x**

SUMMARY . . . . . **xi**

I INTRODUCTION **1**

1.1 Motivation 1

1.2 Overview 3

1.3 Thesis Statement 6

1.4 Contributions 6

II BACKGROUND **8**

2.1 Concurrency Bugs 8

2.1.1 Notation 8

2.1.2 Data Race 8

2.1.3 Memory-Access Patterns 9

2.1.4 Order Violation 9

2.1.5 Single-Variable Atomicity Violation 10

2.1.6 Multi-Variable Atomicity Violation 11

2.2 Concurrency Bug Detectors 12

III FAULT LOCALIZATION FOR SINGLE-VARIABLE CONCURRENCY BUGS **15**

3.1 Introduction 15

3.2 Technique 17

3.2.1 Step 1: Online Pattern Identification 17

3.2.2 Step 2: Pattern Suspiciousness Ranking 20

3.3 Evaluation 22

3.3.1 Implementation 22

3.3.2 Empirical Setup . . . 23
3.3.3 Evaluation of Effectiveness . . . 24
3.4 Related Work . . . 26
3.5 Summary . . . 28
IV FAULT LOCALIZATION FOR MULTIPLE-VARIABLE CONCURRENCY BUGS . . . 29
4.1 Introduction . . . 29
4.2 Technique . . . 31
4.2.1 Step 1: Collect Pairs from Executions . . . 32
4.2.2 Step 2: Combine Pairs into Patterns . . . 38
4.2.3 Example . . . 39
4.2.4 Step 3: Rank Patterns . . . 40
4.3 Evaluation . . . 41
4.3.1 Implementation . . . 41
4.3.2 Empirical Setup . . . 43
4.3.3 Effectiveness of UNICORN . . . 44
4.3.4 Threats to Validity . . . 46
4.4 Related Work . . . 46
4.5 Summary . . . 49
V FAULT EXPLANATION FOR CONCURRENCY BUGS . . . 50
5.1 Introduction . . . 50
5.2 Problems in Existing Techniques . . . 52
5.2.1 Example . . . 53
5.2.2 Problems . . . 54
5.2.3 Challenges . . . 56
5.3 Technique . . . 57
5.3.1 Step 1: Localize Problematic Patterns . . . 58
5.3.2 Step 2: Cluster Failing Executions . . . 59
5.3.3 Step 3: Reconstruct Bug Context . . . 62

5.4 Evaluation 66
5.4.1 Implementation 66
5.4.2 Empirical Setup 66
5.4.3 Study 1: Handling Multiple Bugs 68
5.4.4 Study 2: Reconstructing Bug Context 70
5.4.5 Threats to Validity 73
5.5 Related Work 73
5.6 Summary 75
VI EMPIRICAL USER STUDY OF DEBUGGING TECHNIQUES 77
6.1 Introduction 77
6.2 Related Work 78
6.2.1 Empirical User Studies for Sequential Bugs 78
6.2.2 Visualization Tools for Concurrent Software 79
6.2.3 Empirical User Studies for Concurrency Bugs 79
6.3 Goal and Hypotheses 80
6.4 Experimental Protocol 81
6.4.1 Program Subjects 81
6.4.2 Debugging Tools 83
6.4.3 Tasks 87
6.4.4 Participants 88
6.4.5 Study Design 88
6.4.6 Procedure 88
6.4.7 Evaluation Method 89
6.4.8 Data Availability 90
6.5 Results 90
6.5.1 Overall Results 90
6.5.2 Results by Tool Preference 94
6.6 Limitations 96

6.7 Discussion 97
6.7.1 Experience on The Tools 98
6.7.2 Improvements 99
6.8 Summary 101
VII CONCLUSION AND FUTURE DIRECTIONS 102
APPENDIX A — SUBJECT PROGRAMS 105
APPENDIX B — SURVEY 107
REFERENCES 113

LIST OF TABLES

1 Problematic memory-access patterns. 9

2 Memory-access patterns of fault-localization techniques. 13

3 Subjects used in evaluating Falcon. 23

4 Effectiveness of Falcon. 25

5 Execution example of a sample program 39

6 Subjects used in evaluating Unicorn. 43

7 Effectiveness of Unicorn. 44

8 Comparison between Unicorn and other techniques 47

9 Example output of Unicorn 55

10 Subjects used in evaluating Griffin. 67

11 Study 1: Handling multiple bugs. 69

12 Study 2: Reconstructing bug context. 71

13 Hypothesis testing results 93

14 Average score by tool preference 95

LIST OF FIGURES

1 Overview of the thesis. 4
2 Order violation 10
3 Single-variable atomicity violation 11
4 Multi-variable atomicity violation 12
5 Overview of UNICORN 32
6 Two windows of UNICORN 35
7 Intuition for pair combination 37
8 Atomicity violation in `Vector` 53
9 Problematic memory accesses with call stacks. 56
10 Overview of GRIFFIN. 58
11 Coverage matrix 59
12 Bug graph 65
13 Eclipse plugin for TRACER 85
14 Eclipse plugin for UNICORN. 86
15 Eclipse plugin for GRIFFIN. 87
16 Overall score distribution 91

SUMMARY

Multicore and Internet cloud systems have been widely adopted in recent years and have resulted in the increased development of concurrent programs. However, concurrency bugs are still difficult to test and debug for at least two reasons. Concurrent programs have large interleaving space, and concurrency bugs involve complex interactions among multiple threads.

Existing testing solutions for concurrency bugs have focused on exposing concurrency bugs in the large interleaving space, but they often do not provide debugging information for developers to understand the bugs. To address the problem, this thesis proposes techniques that help developers in debugging concurrency bugs, particularly for locating the root causes and for understanding them, and presents a set of empirical user studies that evaluates the techniques.

First, this thesis introduces a dynamic fault-localization technique, called Falcon, that locates single-variable concurrency bugs as memory-access patterns. Falcon uses dynamic pattern detection and statistical fault localization to report a ranked list of memory-access patterns for root causes of concurrency bugs. The overall Falcon approach is effective: in an empirical evaluation, we show that Falcon ranks program fragments corresponding to the root-cause of the concurrency bug as "most suspicious" almost always. In principle, such a ranking can save a developer's time by allowing him or her to quickly hone in on the problematic code, rather than having to sort through many reports.

Others have shown that single- and multi-variable bugs cover a high fraction of all concurrency bugs that have been documented in a variety of major open-source packages; thus, being able to detect both is important. Because Falcon is limited

to detecting single-variable bugs, we extend the FALCON technique to handle both single-variable and multi-variable bugs, using a unified technique, called UNICORN. UNICORN uses online memory monitoring and offline memory pattern combination to handle multi-variable concurrency bugs. The overall Unicorn approach is effective in ranking memory-access patterns for single- and multi-variable concurrency bugs.

To further assist developers in understanding concurrency bugs, this thesis presents a fault-explanation technique, called GRIFFIN, that provides more context of the root cause than UNICORN. GRIFFIN reconstructs the root cause of the concurrency bugs by grouping suspicious memory accesses, finding suspicious method locations, and presenting calling stacks along with the buggy interleavings. By providing additional context, the overall GRIFFIN approach can provide more information at a higher-level to the developer, allowing him or her to more readily diagnose complex bugs that may cross file or module boundaries.

Finally, this thesis presents a set of empirical user studies that investigates the effectiveness of the presented techniques. In particular, the studies compare the effectiveness between a state-of-the-art debugging technique and our debugging techniques, UNICORN and GRIFFIN. Among our findings, the user study shows that while the techniques are indistinguishable when the fault is relatively simple, GRIFFIN is most effective for more complex faults. This observation further suggests that there may be a need for a spectrum of tools or interfaces that depend on the complexity of the underlying fault or even the background of the user.

CHAPTER I

INTRODUCTION

1.1 Motivation

Multicore systems have been deployed in all kinds of computing systems from Internet cloud systems to desktops to mobile systems for performance benefits, and thus have resulted in the increased development of concurrent programs for those systems [8]. For example, a survey performed at Microsoft in 2007 on 684 technical staffs revealed that concurrency is prevalent; over 60% of respondents had to deal with concurrency issues, and half of those people do it on at least monthly basis [23]. Furthermore, most concurrency bugs[1] are of high severity; on a severity scale of 1 (most severe) to 4 (least severe), more than 80% of respondents rated concurrency bugs as either 1 or 2. Even worse, concurrency bugs on the deployed systems can result in serious disasters; the oft-cited 2003 Northeastern U.S. electricity blackout, which left 10 million people without power, has been attributed in part to a race condition in a monitoring software with multi-million lines of code [73]. Recently, Nasdaq's Facebook IPO glitch, which occurred because of a race condition, has resulted in a loss of millions of dollars [3]. Thus, it is extremely important for businesses to detect and fix concurrency bugs to avoid such catastrophic losses.

Attempts to address concurrency bugs are estimated to consume enormous cost in industrial software development and maintenance. In the Microsoft survey, cited above, more than half of the respondents of handling concurrency issues had to detect, debug and fix concurrency bugs. Furthermore, on average, developers spend seven days between finding a concurrency bug and applying a fix. Sometimes this duration

[1] We use errors, bugs, and faults interchangeably.

lasts several months. In aggregate, debugging cost of concurrency bugs account for "thousands of days of work".

Testing and debugging concurrent software can be even more challenging than for sequential programs, for at least two reasons [56]. First, concurrent programs exhibit more nondeterministic behavior, which can make it difficult to even expose the fault. Nondeterministic bugs always rank as the most common and difficult errors in numerous studies, independent of the programming model [14, 23, 86]. Second, concurrent faults typically involve changes in program state due to particular interleavings of multiple threads of execution, making them difficult to find and understand. These faults most frequently manifest as data races, atomicity violations, and order violations, which are consistently ranked as the most common and difficult source of concurrency faults [50, 86].

To deal with the challenges, many testing-based approaches have focused on exposing concurrency bugs in the large interleaving space. One type of approaches identifies buggy interleavings that lead to concurrency bugs by exploring the interleaving space with systematic or random strategies [17, 57, 58, 65, 70, 82, 89]. Another type of approaches detects specific buggy interleavings that lead to one type of concurrency bugs, such as data races or atomicity violations [11, 19–22, 57, 60, 61, 63, 77, 81, 82, 90]. These techniques help developers in identifying the existence of concurrency bugs, but often do not provide much debugging information to understand and fix the bugs. For instance, the interleaving exploration approaches find the buggy interleaving, but do not provide the root cause or the type of the concurrency bug. For another instance, the bug-directed approaches often report many false positives or benign results,[2] and so developers may need more context to determine whether the results contain the real bug.

[2] A *benign* data race is an intentional data race whose existence does not affect the correctness of the program.

While testing-based approaches have been well researched, little attention has been given to debugging concurrency bugs despite its importance. This is because debugging concurrency bugs are difficult and costly. For example, Mozilla developers spent nearly two months to completely understand and fix a concurrency bug, even though they had a test case to the concurrency bug with the specific interleaving [31]. Another recent survey reveals that concurrency bugs are the most difficult types of software bugs to write a correct patch for, and developers need more guidance for understanding and fixing them [93].

Thus, this thesis focuses on developing techniques that assists developers for debugging concurrency bugs. In the following overview, we present the steps consisting of the debugging process and our research techniques, each of which helps the debugging steps.

1.2 Overview

Debugging is a software process that involves several steps: finding a bug, understanding it, and fixing it with a patch [71, 96]. The first step, *fault localization*, is an activity, where the programmer finds the location of the bug in the program. The second step, *fault understanding*, involves understanding of the root cause of the fault. The final step, *fault correction*, is to determine how to modify the code to fix the fault.

For the thesis, we focus on the fault localization and understanding in the debugging steps. We do not present techniques for fault correction, but we discuss existing fault-correction techniques for concurrent programs in the related work in Chapter 5 and in the future work in Chapter 7.

Figure 1 illustrates the overview of this thesis with a focus on our research steps and the debugging steps. For our research steps 1 and 2, we develop techniques that address the fault-localization problem by locating the root cause of single-variable

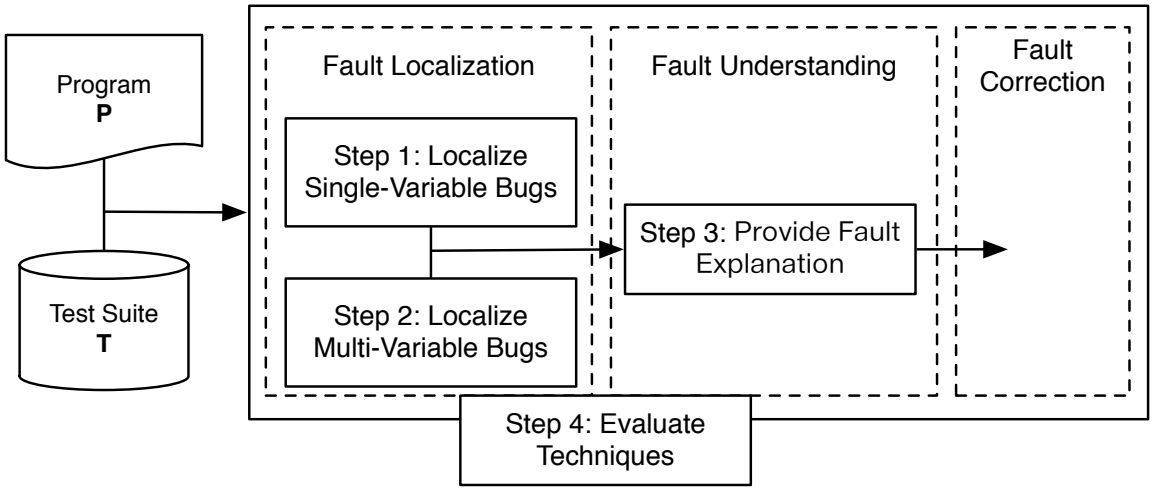


Figure 1: Overview of the thesis.

and multi-variable concurrency bugs using memory-access patterns. Then, for our research step 3, we develop a technique that addresses the fault-understanding problem by presenting the explanation of the fault with context information. Finally, for our research step 4, we evaluate the developed techniques with a state-of-the-art tool through empirical user studies.

Step 1: Fault-localization for single-variable concurrency bugs Arguably, order and atomicity violations are the most important types of non-deadlock concurrency bugs, which consist of more than 90% of non-deadlock concurrency bugs [50]. Existing techniques focused on finding one type of concurrency bugs, such as data races or atomicity violations. A recent trend is to use statistical analysis and report suspicious memory accesses to handle multiple types of bugs. However, these techniques do not provide information of the bug in detail, e.g., the bug type. To address such limitations, we develop a fault-localization technique, called Falcon, that handle both order and single-variable atomicity violations with dynamic pattern detection. In particular, the technique dynamically monitors memory-access patterns for order and atomicity violations, records outcomes of passing and failing executions, and associates the outcomes with patterns to compute suspiciousness of

patterns. Finally, the technique reports the ranked memory-access patterns, and thus, the developer can inspect highly ranked patterns as concurrency bugs.

Step 2: Fault-localization for both single-variable and multi-variable concurrency bugs FALCON is limited to diagnose single-variable non-deadlock concurrency bugs, where 34% of non-deadlock concurrency bugs are multi-variable concurrency bugs [50]. Although there exists bug detectors that handle multi-variable bugs, their coverage is limited to atomicity violations [53, 59]. To address the limitations, we develop a unified technique, called UNICORN, that finds the locations of concurrency bugs involving both single and multiple variables. Because dynamically detecting patterns for multiple variables is costly, this technique dynamically collects pairs of memory accesses. Then, it combines the pairs into patterns for both single- and multi-variable concurrency bugs. Finally, like FALCON, it applies statistical fault localization to report ranked memory-access patterns for bug candidates.

Step 3: Fault-explanation for concurrency bugs A recent study has shown that statistical fault-localization techniques for sequential programs may not provide enough information for developers to understand the bug, and so the techniques need to provide more context of the bugs, such as clustering of the results, to improve the understanding [71]. We found similar problems for statistical fault-localization techniques for concurrent programs, such as UNICORN. For example, the techniques provide too many repetitive memory accesses without any clustering and they lack calling context to the memory accesses. To address the problems, we develop a fault-explanation technique, called GRIFFIN, that explains concurrency bugs with context information. This technique inputs memory accesses with calling context from UNICORN, clusters them based on similar failures, and finally, provides a bug graph, which includes clustered memory access, calling contexts, and suspicious methods.

Step 4: User studies for evaluating the techniques Several fault-localization

techniques for concurrency bugs have been proposed recently [52, 54, 66–69], but none of them have been empirically evaluated to determine whether they really help developers in understanding concurrency bugs. We carry out a set of empirical user studies to investigate the usefulness of the developed automated debugging techniques. In particular, we choose a baseline debugging technique used in industry and compare it with our techniques. We implement the techniques in Eclipse plugin tools and let the developers use the tools for debugging concurrent programs. We observe their activities during the entire debugging process and analyze the results to estimate whether our automated debugging tools are effective to developers.

1.3 Thesis Statement

The thesis of this dissertation is that dynamic fault-localization techniques can assist developers in locating and understanding non-deadlock concurrency bugs by providing memory-access patterns, calling context, and suspicious methods.

1.4 Contributions

This dissertation makes the following contributions.

- Two fault-localization techniques (FALCON and UNICORN) that handle concurrency bugs involving both single and multiple variables using statistical fault-localization. Unlike existing fault-localization techniques, these new techniques have several benefits. First, they suggest more buggy results with higher rank so that developers can focus on the real root cause from benign results. Second, they detect all important types of non-deadlock concurrency bugs, i.e., order and atomicity violations [50] within a single tool, and thus developers do not have to use multiple tools to detect these two types of bugs. Our empirical studies show that FALCON and UNICORN are effective in ranking memory accesses responsible for the bugs at rank 1 or 2.

- A fault-explanation technique (GRIFFIN) that explains concurrency bugs with calling context, suspicious method, and groups of memory accesses. Unlike existing fault-localization techniques that present raw-memory accesses, the new technique presents the bug with a bug graph that reconstructs the buggy interleaving with calling context so that developers can understand the bug easily especially for a harder task. Our empirical studies show that GRIFFIN is effective in providing explanations in several ways: clustering memory accesses for the same bug and pinpointing suspicious method locations.

- A set of empirical user studies that evaluates the effectiveness of the suggested automated debugging techniques for human developers. The empirical results show that GRIFFIN improves understanding of concurrency bugs for hard tasks. The research implications include improvements for the tools and future research directions for debugging concurrency bugs.

- A toolset that implements the techniques that assist debugging concurrency bugs,[3] and a compilation of concurrency bugs we used for our subjects.[4] These tools can be utilized in multiple ways. First, these tools can be used in a classroom for students to learn how concurrency bugs work and how to diagnose concurrency bugs. Second, the code in the toolset can be integrated with existing debuggers, so that developers can use the tool to locate and understand concurrency bugs. Finally, other researchers can build their techniques on top of our implementations and test their tools against ours by using the bugs.

[3] `http://www.cc.gatech.edu/~sangminp/griffin/`
[4] `http://www.cc.gatech.edu/~sangminp/bugs/`

CHAPTER II

BACKGROUND

This section presents background information that enables understanding our approach. Section 2.1 discusses concurrency bugs, and Section 2.2 discusses existing concurrency bug detectors.

2.1 Concurrency Bugs

We begin by introducing our formal notation and then defining the key concurrency violations of interest in this thesis: atomicity violations and order violations.

2.1.1 Notation

We denote a memory access to a shared variable by $b_{t,s}(x)$: b is the memory access type, either read (R) or write (W); t is the thread that performs the access; s is the program statement containing the access; and x is the shared variable. For example, $\mathrm{R}_{1,S_1}(x)$ represents a read access to shared variable x in statement S_1 of thread 1.

2.1.2 Data Race

A *data race* occurs when two or more threads access a shared memory location, where at least one of the accesses is a write, and there is no locking to synchronize the accesses. For example, any of the pairs R_1-W_2, W_1-R_2, W_1-W_2 are, in the absence of synchronization, data races. As is well-known, a data race does not always imply a fault. For example, barriers, flag synchronization, and producer-consumer queues are common concurrency constructs that are implemented with deliberate data races [51]. Therefore, we do not focus on data race detection in this thesis.

Table 1: Problematic memory-access patterns.

PID	Memory-Access Pattern	Memory-Access Pairs
P1	$R_{1,S_i}(x)$ $W_{2,S_2}(x)$	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$
P2	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$
P3	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$
P4	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$ $R_{1,S_k}(x)$	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_j}(x)$ $R_{1,S_k}(x)$
P5	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$ $R_{1,S_k}(x)$	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_j}(x)$ $R_{1,S_k}(x)$
P6	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$ $W_{1,S_k}(x)$	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$, $R_{2,S_j}(x)$ $W_{1,S_k}(x)$
P7	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$ $W_{1,S_k}(x)$	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_j}(x)$ $W_{1,S_k}(x)$
P8	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$ $W_{1,S_k}(x)$	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_j}(x)$ $W_{1,S_k}(x)$
P9	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$ $W_{2,S_k}(y)$ $W_{1,S_l}(y)$	$W_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_k}(y)$ $W_{1,S_l}(y)$
P10	$W_{1,S_i}(x)$ $W_{2,S_j}(y)$ $W_{2,S_k}(x)$ $W_{1,S_l}(y)$	$W_{1,S_i}(x)$ $W_{2,S_k}(x)$, $W_{2,S_j}(y)$ $W_{1,S_l}(y)$
P11	$W_{1,S_i}(x)$ $W_{2,S_j}(y)$ $W_{1,S_k}(y)$ $W_{2,S_l}(x)$	$W_{1,S_i}(x)$ $W_{2,S_l}(x)$, $W_{2,S_j}(y)$ $W_{1,S_k}(y)$
P12	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$ $R_{2,S_k}(y)$ $W_{1,S_l}(y)$	$W_{1,S_i}(x)$ $R_{2,S_j}(x)$, $R_{2,S_k}(y)$ $W_{1,S_l}(y)$
P13	$W_{1,S_i}(x)$ $R_{2,S_j}(y)$ $R_{2,S_k}(x)$ $W_{1,S_l}(y)$	$W_{1,S_i}(x)$ $R_{2,S_k}(x)$, $R_{2,S_j}(y)$ $W_{1,S_l}(y)$
P14	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$ $W_{2,S_k}(y)$ $R_{1,S_l}(y)$	$R_{1,S_i}(x)$ $W_{2,S_j}(x)$, $W_{2,S_k}(y)$ $R_{1,S_l}(y)$
P15	$R_{1,S_i}(x)$ $W_{2,S_j}(y)$ $W_{2,S_k}(x)$ $R_{1,S_l}(y)$	$R_{1,S_i}(x)$ $W_{2,S_k}(x)$, $W_{2,S_j}(y)$ $R_{1,S_l}(y)$
P16	$R_{1,S_i}(x)$ $W_{2,S_j}(y)$ $R_{1,S_k}(y)$ $W_{2,S_l}(x)$	$R_{1,S_i}(x)$ $W_{2,S_l}(x)$, $W_{2,S_j}(y)$ $R_{1,S_k}(y)$
P17	$W_{1,S_i}(x)$ $R_{2,S_j}(y)$ $W_{1,S_k}(y)$ $R_{2,S_l}(x)$	$W_{1,S_i}(x)$ $R_{2,S_l}(x)$, $R_{2,S_j}(y)$ $W_{1,S_k}(y)$

2.1.3 Memory-Access Patterns

Table 1 lists problematic memory-access patterns that represent the concurrency bugs in which we are interested. The first column shows the pattern ID. The second column shows the memory-access pattern using the shared-variable notation, $b_{t,s}(x)$. We discuss the details of the patterns in Sections 2.1.4 through 2.1.6. The third column shows the memory-access pairs that constitute the memory-access pattern. Note that the memory-access patterns are always decomposed into one or two memory-access pairs.

2.1.4 Order Violation

An *order violation* occurs when threads execute in an unintended order, leading to incorrect program behavior. An order violation manifests as a pattern consisting of two sequential thread accesses to a shared-memory location where at least one of the accesses is a write. See Patterns P1 to P3 in Table 1. Note that the pattern of access

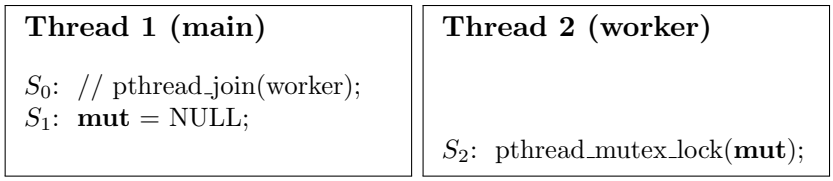


Figure 2: Order violation (extracted from `PBZip2`).

is a race condition; the term *order violation* signifies that the cause is an incorrect ordering of threads.

Figure 2 gives an example of an order violation. The example consists of two threads, where the main thread (Thread 1) should wait at S_0 until the worker thread (Thread 2) finishes. If the main thread does not wait for the worker thread and deinitializes a shared variable `mut` with a null value at S_1, the program crashes with a null-pointer exception violation at S_2. In the example, an unintended order, W_{1,S_1}(`mut`)-R_{2,S_2}(`mut`) (Pattern P2), is an order violation.

2.1.5 Single-Variable Atomicity Violation

A *single-variable atomicity violation* occurs when an unserializable interleaving pattern involving a single variable leads to unintended program behavior. An unserializable interleaving pattern is a sequence of concurrent memory accesses whose resulting state is not the same as that of sequential memory accesses. Patterns P4 to P8 in Table 1 are unserializable interleaving patterns involving a single variable [88]. Vaziri, Tip, and Dolby [88] proved that the unserializable interleaving patterns (P4 to P17 in Table 1) are complete, which means that if an execution conforms to the patterns, the execution is not serializable.

Figure 3 shows an example of a single-variable atomicity violation. The program has two threads. Thread 1 closes an old file and creates a new file, during which `log_type` is temporarily set to CLOSED at S_1 and set to the original status at S_3. Thread 2 records a transaction into a log if `log_type` is not a CLOSED status at S_2. If

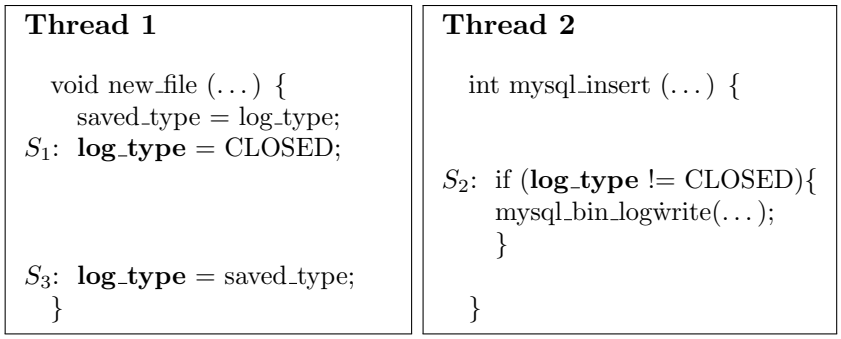


Figure 3: Single-variable atomicity violation (extracted from `Mysql-791`).

`log_type` at S_2 reads a CLOSED status, which is set at S_1, Thread 2 mistakenly misses recording a transaction. Here, the interleaving, W_{1,S_1}(`log_type`)-R_{2,S_2}(`log_type`)-W_{1,S_3}(`log_type`) (Pattern P6 in Table 1), is a single-variable atomicity violation.

2.1.6 Multi-Variable Atomicity Violation

A *multi-variable atomicity violation* occurs when an unserializable interleaving pattern involving multiple variables leads to unintended program behavior. Patterns P9 to P17 in Table 1 are unserializable interleaving patterns involving multiple variables [88].

Figure 4 shows an example of a multi-variable violation. The program has two variables, `TABLE` and `LOG`. The program needs to maintain the invariant that `LOG` records updates to `TABLE` in the order in which those updates occurred. For instance, if an entry is inserted into `TABLE`, the program should immediately log the transaction in `LOG`. Note that `TABLE` and `LOG` are individually protected by locks, but the two operations together are not. Consequently, the interleaving, W_{1,S_1}(`TABLE`)-W_{2,S_2}(`TABLE`)-W_{2,S_3}(`LOG`)-W_{1,S_4}(`LOG`) (Pattern P9), causes `TABLE` and `LOG` to become out-of-sync.

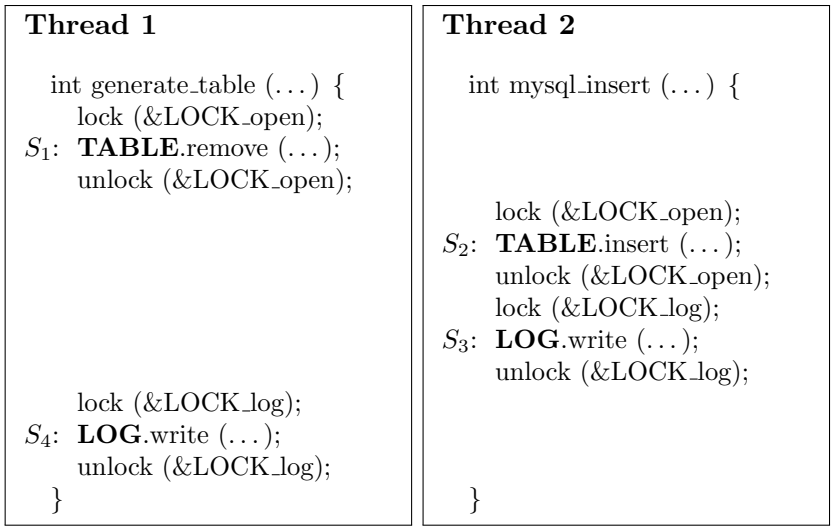


Figure 4: Multi-variable atomicity violation (extracted from `Mysql-169`).

2.2 Concurrency Bug Detectors

Many different types of concurrency bug detectors have been developed to test concurrent programs and debug concurrency bugs. These detectors use static and dynamic approaches to precisely diagnose one type of bugs, such as data races [19, 57, 60, 61, 63, 77, 81, 82] or atomicity violations [11, 20–22, 90]. For our illustrative examples in Figures 2 to 4, data race detectors can detect the order violation in Figure 2 and the single-variable atomicity violation in Figure 3. However, they will miss the multi-variable atomicity violation in Figure 4 because the shared variables are protected by common locks, `LOCK_open` and `LOCK_log`. Atomicity violation detectors can find single- and multi-variable atomicity violations in Figures 3 and 4.

Instead of focusing on the details of the detectors for one type of bugs, we discuss three recent fault-localization-based detectors for concurrent programs [34,54,83] that find all the bugs in Figures 2 to 4. We focus on how each technique presents its bug reports.

All three techniques collect memory accesses between threads during program executions and output a set of memory accesses ranked by suspiciousness. Table 2 shows

Table 2: Memory-access patterns of fault-localization techniques.

Technique	Memory-access pattern		
	Pattern	Memory accesses	Additional information
CCI [34]	W	S_4	tag: thread-remote
DefUse [83]	WW	$S_3 \rightarrow S_4$	
Recon [54]	RWRRW	$S'_3 \rightarrow S_3 \rightarrow S''_3 \rightarrow S'''_3 \rightarrow S_4$	most suspicious: $S_3 \rightarrow S_4$

the comparison of memory-access patterns of the four techniques for the example in Figure 4. The first column lists the techniques. The second column shows the memory-access patterns for each technique designated by the memory-access types (i.e., read (R) or write (W)). The third column shows the memory accesses with respect to the pattern that are ranked 1st for each technique. The fourth column shows the additional information if it is provided by the technique.

CCI [34] detects concurrency bugs using sampling and statistical methods. The technique samples memory-access locations, records each access along with a tag that indicates whether the previous access is thread-local or thread-remote, and records the execution output as passing or failing. Then, the technique computes the suspiciousness of memory-access locations using the statistics of execution output, and outputs a ranked list of the memory-access locations along with their associated tags. To illustrate, consider the second row in Table 2. CCI identifies the read access in S_4 of Figure 4 as the most suspicious location. CCI also reports a tag indicating that the write access is thread-remote.

DefUse [83] detects concurrency bugs that violate definition-use (i.e., write and read) invariants. The technique collects definition-use pairs between two threads in passing executions. Then, the technique finds the definition-use pairs in the failing executions that are not in the set of pairs in passing executions. To illustrate, consider the third row of Table 2. DefUse reports the bug in Figure 4 as the definition-use pair of `LOG` appearing in S_3 and S_4.

Recon [54] detects concurrency bugs using a form of a memory-access graph, called

a context-aware communication graph. The graph shows five consecutive accesses of a memory location regardless of the memory-access type, and computes the most suspicious context change among the five accesses. Recon collects these memory accesses as graphs in multiple program executions, and ranks the graphs. To illustrate, consider the fourth row in Table 2. Recon reports these five consecutive accesses as a graph; for ease of presentation, we list them instead of showing them as a graph. Because Recon records all consecutive dynamic accesses near the bug without filtering non-crucial accesses to the bug, its output contains lines, such as S_3', S_3'' and S_3''', that do not appear in Figure 4. Recon also reports additional information: that $S_3 \rightarrow S_4$ is the most suspicious thread-context edge among the five accesses.

Note that, to identify the multi-variable atomicity violation, techniques should identify at least four memory locations: entry into an atomic region, two interferences, and exit out of the atomic region. In this example, the locations are accesses of `TABLE` and `LOG` in $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4$. However, all three techniques mentioned above report only partial (i.e., one or two) memory-access locations that constitute the atomicity violation. We develop and present our techniques to report all critical accesses consisting of the violations in later Chapters.

CHAPTER III

FAULT LOCALIZATION FOR SINGLE-VARIABLE CONCURRENCY BUGS

3.1 Introduction

Numerous efforts have discovered principles and methods to pinpoint these most frequent concurrent faults. These detectors use static and dynamic approaches to precisely diagnose one type of bugs, such as data races [19, 57, 60, 61, 63, 77, 81, 82] or atomicity violations [11, 20–22, 90]. However, benign data races are common and those data race detectors can yield high false-positive rates [50]. In addition, atomicity violation detectors typically rely on the developer to explicitly annotate atomic regions for subsequent static or dynamic verification by a tool [20, 22].

To relieve this annotation burden, a recent trend is to apply dynamic pattern analysis [25, 51]. The technique characterizes faults by likely interleaved sequences of operations, and then searches for these patterns in an execution. A pattern-based approach can in principle be applied to both atomicity and order violations, although existing methods have thus far considered only atomicity [50]. Furthermore, current methods may report many patterns, only some of which might directly identify the fault. These methods do not presently have any way to rank or prioritize the patterns.

Many existing effective ranking techniques for fault localization are based on code coverage. Examples include prior work on Tarantula for sequential programs [38], and recent work for statement and expression (predicate) ranking for concurrent programs [34, 52]. These methods work by associating the number of occurrences of a target coverage criterion with passing and failing executions, and use these data to compute suspiciousness scores. However, thus far this approach has not been applied

to concurrency patterns.

We propose a new pattern-based dynamic analysis technique for fault localization in concurrent programs that combines pattern identification with statistical rankings of suspiciousness of those patterns. We apply our technique to both single-variable atomicity and order violations.[1] During testing, our technique detects access patterns from actual program executions, which either pass or fail. For each pattern, the technique uses the pass/fail statistics to compute a measure of suspiciousness that is used to rank all occurring patterns, in the spirit of Tarantula in the sequential case [38]. We also describe FALCON, a prototype implementation of the technique in Java, that is designed to have reasonable storage and execution time overheads, so that it may be deployed in realistic testing environments. We used FALCON to perform an empirical study on several Java benchmarks. The empirical study shows that the technique can effectively localize the bug locations for our subjects.

The technique has several advantages over existing tools. First, the technique captures not only order violations but also atomicity violations: existing tools focus only on either of the bugs. Second, the technique reports the real violation patterns with high priority, unlike other techniques that report benign and real violations without priority. In short, our approach provides the same benefits of prior dynamic pattern analysis methods, and contributes an explicit prioritized ranking of those patterns to guide the developer toward the most likely cause of a fault.

The main contributions of the work are summarized as follows:

- To the best of our knowledge, our approach is the first to localize malicious interleaving patterns in concurrent programs. The aim is to help the developer more quickly identify the source of a concurrency fault.
- Our technique detects both single-variable atomicity and order violations. In

[1] According to a study of concurrency bug characteristics, order and atomicity violations are the most important classes of non-deadlock concurrency bugs [50]. However, note that we only handle single-variable atomicity violations and may miss multi-variable atomicity violations.

particular, we believe our work is the first to effectively identify order violations.

- We have implemented this technique in a Java-based toolset, called FALCON, which can be applied to any concurrent program with test cases.
- We evaluate FALCON experimentally, and show that it is effective in localizing concurrency faults in our subjects.

3.2 Technique

Our technique for identifying these non-deadlock concurrency bugs for Java threads consists of two main steps. In Step 1, the technique monitors shared-memory accesses online, detecting and recording *patterns* of such accesses that correspond to problematic interleaving patterns in Table 1.[2] Step 1 associates these patterns with test cases and pass/fail results of the executions. In Step 2, our technique applies statistical analysis to the results of Step 1 to compute a *suspiciousness value* for each detected pattern, as described in Section 3.2.2. Using these suspiciousness values, Step 2 ranks the patterns from most to least suspicious, and presents this ordered list to the user. Section 3.2 describes these steps in detail.

3.2.1 Step 1: Online Pattern Identification

Step 1 of the technique identifies problematic memory-access patterns (P1 to P8 in Table 1) during the program's execution. Like other fault-localization techniques [6, 38, 39, 42], our technique records program entities and subsequently associates them with passing and failing runs.

However, our technique differs from prior fault-localization work in two ways. First, instead of running the program with many test cases, our technique runs the program many times (i.e., k times) with the same test case. The program is non-deterministic; thus, different and possibly faulty interleavings of access to shared

[2]Recall that these patterns are associated with atomicity violations and order violations, respectively.

variables can occur in different executions of the same test cases. (We also apply random delays (or *irritators*) to increase the likelihood of different interleavings [17, 89].) Second, instead of gathering coverage of simple entities, such as statements, branches, or predicates, our technique tracks patterns (i.e., sequences of shared variable references).

An instrumented version of the program, which we call P', executes each test case k times. During these executions, our technique uses a *fixed-sized sliding-window* mechanism to identify patterns. For each execution, our technique associates patterns with program-execution behavior: passing (i.e., the program behaved as expected) or failing (i.e., the program exhibited unexpected behavior). After all k executions, the set of suspicious patterns and the number of passing and failing executions associated with each pattern is passed to Step 2 of the algorithm.

Windowing scheme and update policy. As P' executes with a test case, it maintains a set of fixed-size windows that store memory-access information. There is one window for each shared-memory location. When tracking patterns, using a fixed-size data structure for each memory location reduces the time and storage overheads compared to recording all shared-memory accesses. than maintaining all shared memory accesses. With fixed-size windows, the storage overhead grows with the number of shared variables rather than the number of memory accesses.

When any thread references the variable, our technique updates its associated window. Initially, the window is empty so our technique always records the first reference. If a new reference occurs in a different thread from the previously recorded reference—a *thread-remote* access—our technique records the new reference in the next slot. Otherwise, the threads are the same—a *thread-local* access—and our technique replaces the previous reference. One exception to this replacement is when the new reference is a read and the last reference was a write, in which case we keep the write. That is, we heuristically prefer writes, largely because we know that both

Algorithm 1: GatherPatterns.

Input : m: shared memory location
b: memory access type
t: thread ID
s: memory access location
Pt: current set of patterns (initially null)

Output: Pt: updated set of patterns

```
1  if m does not yet have any window then
2  |  w=createWindow ();
3  |  w.insert(b,t,s);
4  |  registerWindow (w,m);
5  else
6  |  w=getWindow (m);
7  |  (b2,t2,s2)=w.getLastAccess();
8  |  if t=t2 then
9  |  |  w.update(b,s);
10 |  else
11 |  |  if w is full then
12 |  |  |  Pt+=getPatterns (w);
13 |  |  |  w=slideWindow (w);
14 |  |  end
15 |  |  w.insert(b,t,s);
16 |  end
17 end
18 return Pt;
```

order and atomicity violations require at least one write.

This scheme is approximate in the sense that it may prematurely evict references that are part of some pattern, owing to the limited capacity of the window. Tuning the window size allows our technique to trade accuracy for time and storage overhead.

The online pattern gathering algorithm. Our overall pattern-collection algorithm, shown in Algorithm 1, is invoked whenever there is a new reference (b, t, s) to shared-memory location m, and that also includes the window update policy described above. Because this algorithm is gathering patterns online, it assumes there is some current set of patterns, Pt, and updates this set.

The algorithm first checks whether a window exists for m (line 1). If not, it creates one (lines 2–4). Otherwise, it retrieves the window w from a global table, extracts the

last access (lines 6–7), and updates the window (lines 8–17) using the window-update policy described previously.

If, during the window update, the algorithm discovers that the window is full (line 11), then it scans the window for patterns (lines 12), and finally slides the window (line 13). When extracting patterns, the algorithm checks the window for all of the interleaving patterns for atomicity violation in Table 1, where the first access in the pattern indicates the oldest slot. If there is no patterns for atomicity violation, the algorithm checks the window for patterns for order violation in Table 1. That is, the algorithm does not doubly count a pair (pattern for order violation) that is already detected as a triplet (pattern for atomicity violation).

Theoretically, if we want to guarantee that our technique does not miss any patterns, we can argue bounds on the necessary window size as follows. A trivial lower bound on the window size for detecting the patterns for atomicity violation is the maximum length of any pattern. For the patterns in this study (Table 1), the longest pattern has 3 references, so a lower bound on window size for our patterns is 3.

However, the upper bound should also be proportional to the number of threads (without any compression). To see this bound, suppose we wish only to gather patterns of the form R_i-W_j-R_i, and that there are n threads of execution. Consider an actual execution with the reference stream, R_1-W_2-W_3-$\cdots$-W_n-R_1. Clearly, we need at least $O(n)$ slots to capture all $n-1$ patterns of the form R_1-W_j-R_1. Thus, we might expect that as n increases, we need to increase the window size accordingly. Moreover, we might expect that this window size might need to grow by as much as $O(n^2)$ in the worst possible case, since the $O(n)$ bound applies to just a single thread.

3.2.2 Step 2: Pattern Suspiciousness Ranking

Step 2 of the technique uses the results of Step 1—the gathered patterns and their association with passing and failing executions—and computes a *suspiciousness score*

using statistical analysis.

Basic approach: Suspiciousness scores for patterns. There is a body of research on statistical analysis for fault localization for sequential or deterministic programs [6, 38, 42]. These approaches assume that entities (e.g., statements, branches, and predicates) executed more often by failing executions than passing executions are more suspect. Thus, they associate each entity with a suspiciousness score that reflects this hypothesis. For example, Tarantula uses the following formula, where s is a statement, $\%passed(s)$ is the percentage of the passing test executions that execute s, and $\%failed(s)$ is the percentage of the failing test executions that execute s [38]:

$$\text{suspiciousness}_T(s) = \frac{\%\text{failed}(s)}{\%\text{failed}(s) + \%\text{passed}(s)} \tag{1}$$

For concurrent programs, we can apply the same methodology, including the Tarantula formula, to score *patterns*. This approach works in a reasonable way most of the time, but sometimes produces unexpected suspiciousness values.

The problem arises from the non-determinism inherent in concurrent programs. It is possible that a pattern occurs in only one failing execution and no passing executions, but is not related to the real fault in the program. In this case, the Tarantula formula (1) gives this pattern a suspiciousness value of 1—the highest suspiciousness value. To account for this case, the formula should assign a higher score to patterns that appear more frequently in failing cases.

Our scoring approach: Jaccard Index. The *Jaccard index* addresses this weighting issue by comparing the similarity of the passing and failing sets [6]. We use this measure in our technique. For a pattern s, where $passed(s)$ is the number of passing executions in which we observe s, $failed(s)$ is the number of failing executions, and $totalfailed$ is the number of total failures, we use the following score:

$$\text{suspiciousness}_J(s) = \frac{\text{failed}(s)}{\text{totalfailed} + \text{passed}(s)} \tag{2}$$

3.3 Evaluation

We implemented a prototype of our fault-localization technique in a tool, called FALCON. In this section, we empirically evaluate FALCON by assessing the effectiveness of our ranking algorithm (Section 3.3.3).

3.3.1 Implementation

We implemented FALCON in Java. Not counting other software that it uses, FALCON consists of 7224 lines of code.

The first main component of FALCON is its instrumentation and monitoring capabilities. For the instrumentation component of FALCON, we used the Soot Analysis Framework,[3] which analyzes programs in Java bytecode format. FALCON performs a static thread-escape analysis [24] to determine which variables might be shared among multiple threads, and instruments the program to observe and record shared accesses at runtime.

FALCON also instruments methods, to provide detailed stack-trace information in subsequent bug analysis. Moreover, FALCON provides an option to inject artificial-delays that can increase the number of interleavings that occur, thereby increasing the chance of eliciting concurrency bugs [17, 89]. We use this option in our experiments. The FALCON dynamic monitor executes in a separate thread as the instrumented program executes. This monitor dynamically receives memory-access information generated from multiple threads in a non-blocking concurrent queue, which maintains memory accesses in a sequential order. The accesses are obtained from this queue to construct windows for extracting patterns (Section 3.2.1).

The second main component of FALCON computes suspiciousness values for each pattern (Section 3.2.2), and reports the list of ranked suspicious patterns in a text format. Within the FALCON toolset, each suspicious pattern can be represented in

[3] `http://www.sable.mcgill.ca/soot/`

Table 3: Subjects used in evaluating FALCON.

Program		LOC	% Failed	Type of Bug
Contest Benchmarks	Account	155	3%	Atomicity
	AirlinesTickets	95	54%	Atomicity
	BubbleSort2	130	69%	Atomicity
	BufWriter	255	14%	Atomicity
	Lottery	359	43%	Atomicity
	MergeSort	375	84%	Atomicity
	Shop	273	2%	Atomicity
Java Collection	ArrayList	5866	2%	Atomicity
	HashSet	7086	3%	Atomicity
	StringBuffer	1320	3%	Atomicity
	TreeSet	7532	3%	Atomicity
	Vector	709	2%	Atomicity
Miscellaneous	Cache4j	3897	3%	Order
	Hedc	29947	1%	Atomicity
	Philo	110	0%	Atomicity
	RayTracer	1924	14%	Atomicity
	Tsp	720	0%	Atomicity

dotty[4] graphical format.

3.3.2 Empirical Setup

Table 3 describes the set of subject programs we used in our study. The first and second columns list the subject programs, classified into three categories: Contest benchmarks, Java Collection Library, and Miscellaneous (Misc) programs. The third column shows the size of the subject program in lines of code. The fourth column displays the empirically observed failure rate, to give a rough sense of the difficulty of eliciting a fault. The fifth column classifies concurrency violation type for the known bug as either an atomicity or an order violation.

We ran our experiments on a desktop computer with a 2.66 GHz Intel Core 2 Duo processor and 4GB RAM, using Windows Vista and Sun's Java 1.5.

[4] `http://www.graphviz.org/`

3.3.3 Evaluation of Effectiveness

The goal of this study is to investigate how well our technique ranks patterns by determining whether highly ranked patterns correspond to true bugs. To do this, we used the FALCON prototype with artificial delays and window size 5 to get the ranked patterns. We used the benchmark programs with their default number of threads, and executed each program k=100 times.

Columns 2–5 in Table 4 summarize the results of this study for the programs listed in the first column. Column 2 reports the highest observed suspiciousness value; column 3 reports the number of patterns identified; and column 4 reports the number of patterns appearing in at least one failing execution. We only report the number of problematic interleaving patterns, according to the program's violation type (see Table 3). For instance, in Cache4j, the number of patterns indicates the number of patterns for order violation since it contains an order violation; in the other programs, the number of patterns indicates the number of patterns for atomicity violation. Column 5 shows the highest rank of the first pattern found by FALCON that corresponds to a true violation. For example, the Account program has 11 patterns typical of atomicity violations, among which 10 patterns appeared in at least one failing execution, and the highest rank assigned by FALCON to any pattern corresponding to a true bug was 2.

We observe that FALCON is effective for our subjects because it identifies a true bug as either its first or its second ranked pattern. This result implies that a programmer need only look at the first or second pattern reported by FALCON to find an actual bug.

By contrast, other atomicity violation detectors that do not rank and report more patterns, implying potentially more programmer effort to examine the report. Several of these approaches [20, 25] will report the number of patterns shown in Column 3, without additional filtering. The AVIO technique [51] would reduce these patterns,

Table 4: Effectiveness of Falcon.

Program	Suspiciousness	# Patterns	# Patterns in Fail	Rank
Account	0.8	11	10	2
AirlinesTickets	0.33	4	1	1
BubbleSort2	1	4	4	1
BufWriter	0.33	105	71	1
Lottery	0.1	4	4	1
MergeSort	0.9	76	63	1
Shop	0.16	10	2	2
ArrayList	1	1	1	1
HashSet	1	7	3	1
StringBuffer	1	2	1	1
TreeSet	1	9	5	2
Vector	1	1	1	1
Cache4j	0.51	23	12	1
Hedc	0.01	2	2	0
Philo	0	9	0	0
RayTracer	1	14	14	2
Tsp	0	24	0	0

instead reporting the number of patterns shown in Column 5. However, this number of patterns still implies more programmer's effort than with Falcon.

Falcon also works effectively even if the program has multiple concurrency bugs, which is the case with the Contest benchmarks. For the Contest benchmarks, Falcon reports different patterns as the most suspicious pattern from different experiments, but the most suspicious pattern was always a real bug.

There are three special cases in our data. Philo and Tsp did not fail at all during our many runs of the programs, and thus, we cannot report any suspicious patterns for them. (That is, we detect all patterns but all suspiciousness scores will be 0 if there are no failing cases.) Hedc is the only case in which we cannot pinpoint the real bug, because the bug is hidden in the library code. The bug is triggered when a shared object concurrently calls a library method from multiple threads. Because the bug location is in uninstrumented library code, bug detection tools including Falcon

cannot pinpoint the bug location.

3.4 Related Work

Data race detection. Early work focused on static and dynamic approaches to detecting *data races*, which occur when multiple threads perform unsynchronized access to a shared memory location (with at least one write). Static techniques include those based on type systems [19], model checking [57], and program analysis [60]. Dynamic techniques include happens-before (RecPlay [77]) and lockset algorithms (Eraser [81]). There are additional approaches [63, 82], including hybrid analysis [64], that feature improved overheads and reduce the number of false positive reports.

The main drawback of data race detectors is that some races—like those used in barriers, flag synchronization, and producer-consumer queues—are common parallel patterns that *rely* on deliberate but benign races [51]. Programmers are left to sort out benign and problematic cases on their own. In FALCON, we focus on atomicity and order violations (though we can also handle data races) and provide additional information (suspiciousness scores) to help pinpoint a fault's root cause.

Atomicity violation detection. Researchers have suggested that *atomicity* (or *serializability*) is an alternative higher-level property that could be checked to detect concurrency faults. Atomicity violation detectors were first advocated by Flanagan and Qadeer [22]. Atomicity checkers rely on programmer annotations of atomic regions or other constructs, which the checkers can then verify or use to do additional inference. There are numerous static [22], dynamic [20, 21, 90, 92], and hybrid schemes [11]. The main practical drawback of atomicity violation detectors is the need for investigating synchronization keywords to infer atomic regions, which FALCON avoids by using pattern-analysis techniques.

Pattern analysis. FALCON is most closely related to the class of pattern-analysis techniques. These include AVIO, which learns benign data access patterns from

"training" executions; during testing executions, AVIO reports as "malicious" any data access patterns not part of the training set [51]. As discussed in Section 3.3.3, FALCON improves on AVIO by computing suspiciousness scores. This approach prioritizes patterns and mitigates false positive cases. Moreover, it can reduce false negatives, since in AVIO a faulty patterns could appear in both passing and failing executions; in FALCON, our weighted ranking mitigates this effect.

Hammer et al. [25] develop the notion of atomic-set-serializability, an extension of conflict-serializability [90], which can capture atomicity violations with more precision by considering atomic regions. Their tool records data access sequences at runtime. As with AVIO, FALCON improves on the Hammer, et al., technique by ranking patterns.

Bug eliciting techniques for concurrent programs. A drawback of any testing-based approach is that program failures may occur infrequently (if at all). Introducing random delays using irritators can increase the likelihood of a buggy interleaving [17, 89]. More focused (non-random) schemes exist as well. These include schemes that control the scheduler to elicit specific interleavings [82]; run-time monitoring and control of synchronization [65]; and analysis-based methods [70].

Musuvathi et al. published several papers [57, 58] for CHESS model checker, which reduces the interleaving space by bounding the number of preempting context switches. The technique is based on a theorem that limiting context switches only at synchronization points is sufficient to detect all data races in the program [57]. Thus, the tool investigates polynomial time interleaving space, while checking assertion violations, deadlock, livelock, and data races.

In FALCON, we provide the option of introducing random delays, though we did not evaluate it experimentally. In general, we believe bug eliciting methods complement our approach, and combined schemes are possible.

Fault localization. There are a number of fault-localization techniques based on

code coverage, particularly for sequential programs [6, 38, 42]. These methods instrument code predicates (e.g., statements or branches) and check coverage by counting the number of occurrences of the predicates in a number of passing and failing executions. These predicates are then assigned some suspiciousness score. Aside from FALCON, CCI [34] and Bugaboo [52] apply to fault localization. The main distinction of FALCON is that it ranks patterns, which can provide more contextual information than statement or predicate expression ranking as done in CCI or Bugaboo.

3.5 Summary

Our technique for fault localization in concurrent programs combines two promising approaches: (1) dynamic analysis to identify shared memory access patterns associated with single-variable order and atomicity violations, and (2) ranking these patterns statistically, using pass/fail test case data. We believe ours to be the first technique to both report and rank patterns. Our empirical study shows that our implementation, FALCON, is effective in ranking true fault patterns.

CHAPTER IV

FAULT LOCALIZATION FOR MULTIPLE-VARIABLE CONCURRENCY BUGS

4.1 Introduction

Chapter 3 showed that FALCON is effective in locating memory-access patterns for concurrency bugs. However, its bug diagnosis coverage is limited to single-variable concurrency bugs, like other single-variable atomicity violation detectors [51, 70]. Other techniques report the existence of non-deadlock concurrency bugs involving both a single variable and multiple variables [34, 52, 54, 83]. However, these techniques do not report the all memory accesses consisting of the patterns for order and atomicity violations, and thus, provide insufficient information to fully understand the bug.

There are two main limitations of existing techniques. The first limitation is that existing techniques that handle multi-variable atomicity violations and provide sufficient information (e.g., access patterns) are not completely automated. These techniques require annotations of atomic regions [25, 88] or groups of memory locations [53] to find multi-variable problematic patterns only in the specified regions or groups. The second limitation is that there is no unified technique that detects both single- and multi-variable bugs together with sufficient information. A developer can use several existing tools, each of which detects a single type of concurrency bug with sufficient information, to detect all important classes of bugs. However, running several tools separately may increase testing time significantly, and understanding formats of reports from separate tools may require additional effort to find and fix the concurrency bugs.

To address the limitations of existing techniques, we developed UNICORN, a dynamic-pattern-detection technique that handles order, single-variable atomicity, and multi-variable atomicity violations[1] with a unified framework. UNICORN is based on the observation that problematic memory-access patterns representing concurrency bugs consist of several problematic memory-access pairs. UNICORN consists of three steps. In Step 1, UNICORN monitors memory-access pairs using a fixed-sized sliding-window mechanism, and records the program-execution outcome as either passing or failing. In Step 2, UNICORN combines memory-access pairs into problematic memory-access patterns using a second fixed-sized sliding-window mechanism for maintaining pairs. In Step 3, UNICORN computes the suspiciousness of the patterns and orders them in decreasing order of suspiciousness so that developers can quickly find the actual concurrency bugs from the bug report.

UNICORN has several benefits over existing techniques. First, UNICORN is effective in detecting significant classes of bug types, including order violation and single-variable and multi-variable atomicity violations. Thus, UNICORN can provide more information to developers about the types of bugs than other techniques. Second, UNICORN reports patterns in order of suspiciousness, unlike other detectors that report benign and harmful results together without any ordering. Thus, developers save time in finding an actual bug using the bug report because they investigate harmful results before benign results. Finally, UNICORN integrates the detection of several classes of bugs and is implemented as a highly-automated single tool. In contrast, other techniques [34, 83] consist of several separate techniques implemented as single tools, which requires a developer to run all tools and to investigate all reports from the tools to find concurrency bugs that the techniques handle.

We also describe our implementations of UNICORN for both Java and C++, along

[1] According to a study of concurrency bug characteristics, these three types of violations are the most important classes of non-deadlock concurrency bugs [50].

with the results of an empirical study that we performed on a set of subjects to evaluate UNICORN. The study results show that, for our subjects, UNICORN is effective in detecting multi-variable atomicity violations, as well as in detecting order and single-variable atomicity violations.

The main contributions of the work are summarized as follows:

- The presentation of a new technique that handles important classes of concurrency bugs: order violation, and single-variable and multiple-variable atomicity violations. To our knowledge, this is the first technique that targets and detects all these violations together with pattern and rank information.
- A description of implementations of the technique in Java and in C++.
- The results of a set of empirical studies that show the effectiveness of the technique in detecting concurrency bugs for multi-threaded programs. Importantly, problematic patterns that are directly related to the bug in all of our tests appeared at the top of the ranked list, suggesting that UNICORN will help developers locate bugs quickly.

4.2 Technique

UNICORN identifies the problematic memory-access patterns, listed in Table 1, that may cause concurrency bugs. Figure 5 depicts the technique, which inputs a concurrent program P and a test suite T, and consists of three steps. In Step 1, UNICORN executes P with T multiple times, collects memory-access pairs from each execution, and records the program-execution outcomes as passing or failing. In Step 2, for each execution, UNICORN combines the pairs into problematic memory-access patterns (listed in Table 1). In Step 3, UNICORN computes the suspiciousness of the problematic memory-access patterns, and ranks the patterns in decreasing order by suspiciousness. Algorithm 2 gives the details of each step of the technique, and Sections 4.2.1 to 4.2.4 discuss the algorithm in detail. The solution presented in these

Figure 5: Overview of UNICORN; Sections 4.2.1 to 4.2.4 provide details.

sections appears in Reference [68].

4.2.1 Step 1: Collect Pairs from Executions

Step 1 collects memory-access pairs from program executions, and does so *online*. This step inputs a concurrent program P and a test suite T, and outputs memory-access pairs for each execution as $pairs$, and program execution outcome as passing or failing for each execution as *outcomes*.

The algorithm first instruments shared read and write variables in P, and generates an instrumented program P' (line 1). The algorithm uses escape analysis [24] to instrument only shared variables. Then, P' executes T m times to collect $pairs$ and *outcomes* (lines 2–6). For each execution, the algorithm collects problematic memory-access pairs (Patterns P1 to P3) as *pairlist*, and program-execution outcome as *outcome*, determined by T, as passing or failing (line 3). Then, for each execution i, the algorithm associates $pairlist[i]$ with $pairs$ and $outcome[i]$ with $outcomes$, respectively (lines 4–5). After all m executions, the algorithm passes $pairs$ and *outcomes* to Step 2 of the algorithm.

Fault localization for concurrent programs Like other fault-localization techniques, our technique records program entities and subsequently associates them with passing and failing executions. However, our technique differs from prior fault-localization work in two ways. First, instead of running the program with each test case in the test suite only once, our technique runs the program many times (i.e., m times) with each test case in the test suite. The program is non-deterministic, and thus, different and possibly faulty interleavings of accesses to shared variables

Algorithm 2: UNICORN algorithm

Input : P: program, T: test suite, w: window size for pairs, m: number of executions
Output: *rankedPatterns*: a list of patterns ordered by suspiciousness
Data: *pairlist*: a list of pairs in an execution, *orderedPairs*: an ordered list of pairs in an execution, *pairs*: a map of a run id to *pairlist*, *outcomes*: a map of a run id to Pass/Fail, *patternset*: a set of patterns in an execution, *allpatternset*: a set of patterns in all executions, *patterns*: a map of a run id to a set of patterns, *patternsToOutcome*: a map of a pattern with Pass/Fail to occurrence count, *suspmap*: map of a pattern to its suspiciousness value

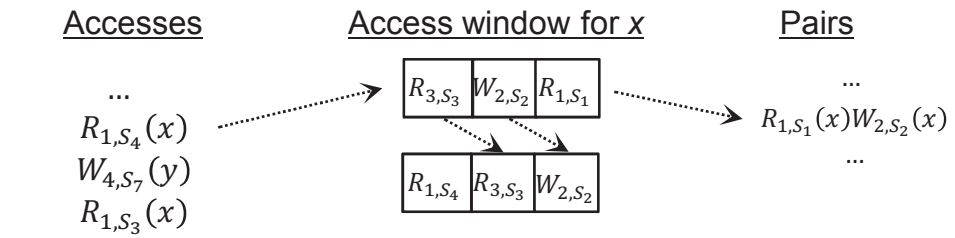
```
   // Step 1 (online): see Section 4.2.1
1  P' = instrument(P)
2  for i ∈ {0..m-1} do
3  |  (pairlist, outcome) = execute(P', T)
4  |  pairs[i] = pairlist
5  |  outcomes[i] = outcome
6  end
   // Step 2 (offline): see Section 4.2.2
7  for i ∈ {0..m-1} do
8  |  patternset = empty
9  |  pairlist = pairs[i]
10 |  for p ∈ pairlist do
11 |  |  patternset.add(p)
12 |  end
13 |  orderedPairs = orderPairs(pairlist)
14 |  n = size(orderedPairs)
15 |  for j ∈ {0..n-w-1} do
16 |  |  for k ∈ {j+1..j+w} do
17 |  |  |  p1 = orderedPairs[j]
18 |  |  |  p2 = orderedPairs[k]
19 |  |  |  if isPattern(p1, p2) then
20 |  |  |  |  pt = makePattern (p1, p2)
21 |  |  |  |  patternset.add(pt)
22 |  |  |  end
23 |  |  end
24 |  end
25 |  patterns[i] = patternset
26 end
   // Step 3 (offline): see Section 4.2.4
27 for i ∈ {0..m-1} do
28 |  outcome = outcomes[i]
29 |  patternset = patterns[i]
30 |  for pt ∈ patternset do
31 |  |  patternsToOutcome[pt][outcomes] += 1
32 |  |  allpatternset.add(p)
33 |  end
34 end
35 for pt ∈ allpatternset do
36 |  pass = patternsToOutcome[pt][Pass]
37 |  fail = patternsToOutcome[pt][Fail]
38 |  susp = computeSusp (pass, fail)
39 |  suspmap[pt] = susp
40 end
41 rankedPatterns = rankPattern(suspmap)
42 return rankedPatterns
```

can occur in different executions of the same test case. Second, instead of gathering coverage of simple entities, such as statements, branches, or predicates, our technique tracks pairs of accesses, and then combines them to patterns (i.e., sequences of shared variable references).

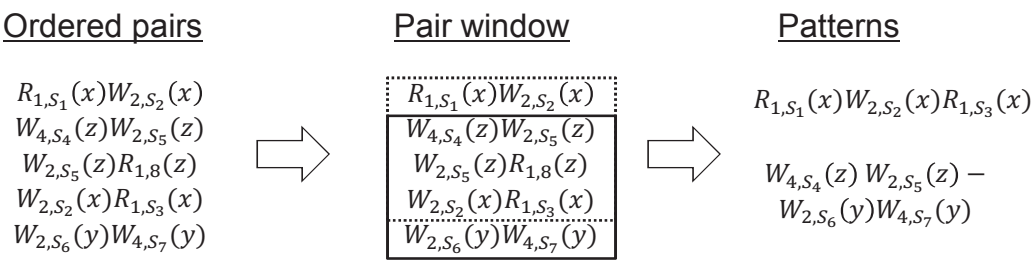
Shared-memory access instrumentation For each shared-memory access encountered during execution of P' (line 3), the algorithm records five kinds of information: the memory location, the static source location, the memory-access type as read or write, the parent thread id, and the global access index of the memory access. The algorithm maintains a global-access index counter during runtime, and thus, the algorithm issues a global-access index for every shared-memory access. The index is used for pattern combination in Step 2.

Access window During execution of P' (line 3), the algorithm uses a fixed-sized sliding-window mechanism to collect problematic memory-access pairs of the shared-memory accesses. Specifically, the algorithm maintains one window (called *access window*) for each shared-memory location. When a shared-memory access occurs, the algorithm recognizes the memory location of the access and adds the access to the window of the same memory location. Inside an access window, there are fixed-sized slots, where each slot is occupied by an access.

The window update policy works as follows. When any access to the memory location occurs, the algorithm updates its associated access window. Initially, the window is empty and the first access is added to a slot in a window. If a new access occurs and the access is from a thread different from the access in the latest slot, the algorithm shifts the accesses in the window and the newest access is added to the latest slot. If a new access occurs and the access is from the same thread as the access in the latest slot, the algorithm replaces the previous access. One exception to this replacement occurs when the new access is a read and the latest access is a write. In this case, the algorithm keeps the write access. The algorithm heuristically



(a): Collect pairs using access window



(b): Collect patterns using pair window

Figure 6: UNICORN uses two windows: (a) collects pairs from shared-memory accesses, and (b) collects patterns from memory-access pairs.

prefers writes because all pairs in Table 1 have at least one write access. When there is an attempt to add an access to a full window, the algorithm finds memory-access pairs involving the oldest access and discards the oldest access.

Figure 6(a) illustrates the sliding-window mechanism for access windows. The left side of the figure (labeled Accesses) shows a serialized representation of three shared-memory accesses from a program execution. The center of the figure (labeled Access window for x) shows access windows of size 3; for simplicity, we show windows only for variable x. The top window shows the contents of the window for x, where the left slot is the newest access ($\mathrm{R}_{3,S_3}(x)$) and the right slot is the oldest access ($\mathrm{R}_{1,S_1}(x)$). When the new shared-memory access ($\mathrm{R}_{1,S_4}(x)$) is executed, the algorithm checks whether the new access and the latest access in the window are from the same or different threads. Because in the example, the accesses are from different threads (Thread 1 and Thread 3), the algorithm adds the new access to the latest slot. However, because the window is full, the algorithm finds pairs involving the

oldest access R_{1,S_1}, generates the pair $\mathrm{R}_{1,S_1}(x)\mathrm{W}_{2,S_2}(x)$, which is shown on the right side of the figure (labeled Pairs), and discards this oldest access. The algorithm then updates the window by adding R_{1,S_4} as the newest access. The updated window is shown at the bottom in the center of Figure 6(a).

The online fixed-sized window scheme enables low time and space overhead. Suppose the number of shared variables is s, and the size of each window is w. Then, the upper bound of space overhead is $O(s \times w)$. Our preliminary study (Study 1 in [67]) suggests, w is effectively a small constant. Other existing techniques [51, 52] also maintain data structures for each shared variable, and thus, they have the same overhead of $O(s)$.

Rationale for online pair identification The key idea behind the UNICORN technique is to collect pairs from executions and combine these pairs offline to get patterns. By doing so, UNICORN can keep the online overhead as low as other techniques [51,52], but can extend its fault localization ability to multi-variable concurrency bugs.

Unlike other techniques [25, 51, 53, 67, 88], UNICORN does not collect patterns online because designing and maintaining online data structures to collect patterns for multi-variable concurrency bugs are complex to implement and may result in significant storage and runtime overheads. Suppose we designed a fixed-sized window technique to identify multi-variable concurrency bug patterns. We could design two types of windows: (1) windows that contains accesses to a single variable to identify patterns for a single variable (P1 to P8 in Table 1); (2) windows that contains accesses of *pairs* of variables, to identity patterns for multiple variables (P9 to P17 in Table 1). In this case, the technique would require overhead of $O(s)$ for the first type of window, and the technique would require overhead of $O(s^2 \times w') = O(s^2)$, where s is the number of memory accesses and w' is the size of the new windows, for the second type of window. Because an algorithm with runtime $O(s^2)$ incurs too much overhead to maintain during program execution, existing techniques use heuristics to limit

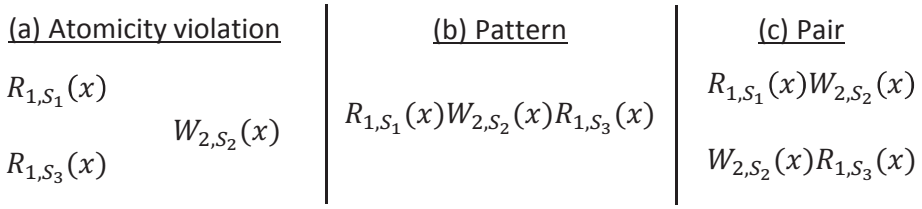


Figure 7: The technique is based on the observation that patterns consist of one or two pairs. (a) is an atomicity violation with $\mathrm{R}_{1,S_1}(x)\mathrm{W}_{2,S_2}(x)\mathrm{R}_{1,S_3}(x)$, which is represented as a pattern in (b), and is represented as two pairs in (c).

the scope of access monitoring (i.e., inside a method or inside an annotated region), and incur large overhead only in some parts of the program execution [25, 53, 88]. By contrast, UNICORN does not limit the scope of access monitoring for identifying patterns for multi-variable concurrency bugs.

The intuition for this key idea is that problematic memory-access patterns can typically be captured by only one or two problematic memory-access pairs. Consider again Table 1. Problematic memory-access patterns consist of two, three, or four memory accesses (second column), and they are represented by one or two memory-accesses pairs (third column). For example, Figure 7(a) shows an atomicity violation with $\mathrm{R}_{1,S_1}(x)\mathrm{W}_{2,S_2}(x)\mathrm{R}_{1,S_3}(x)$, which is represented as a pattern in (b), and is represented as two pairs in (c).

This key idea explains the two advantages of the UNICORN technique. First, UNICORN combines and reports patterns, which are sufficient for detecting multi-variable atomicity violations. Thus, developers can understand both single and multi-variable bugs by inspecting the locations expressed in the patterns. Second, UNICORN maintains reasonable runtime overhead because it monitors only memory-access pairs. UNICORN performs its work offline to get complete bug information by combining pairs into patterns (See details in Step 2).

4.2.2 Step 2: Combine Pairs into Patterns

Step 2 of the algorithm combines memory-access pairs (*pairs*) into problematic memory-access patterns (*patterns*) (line 7), and does so *offline*. For each execution, a list of pairs, *pairlist*, is input and a set of patterns, *patternset*, is output. For patterns consisting of a pair (P1-P3), the algorithm adds all pairs in *pairlist* to *patternset* (lines 10–12). For patterns consisting of two pairs (P4-P17), the algorithm considers the combinations of two pairs in *pairlist* and combines them when they represent a pattern (lines 13–25).

Pair window The algorithm uses a fixed-sized sliding-window mechanism to identify the combinations of the pairs efficiently. Step 1 uses a different window (called *pair window*) for each shared-memory location. However, Step 2 maintains only one window for the entire list of pairs, regardless of shared-memory locations, because Step 2 must combine patterns that involve two memory locations.

The algorithm first sorts *pairlist* in increasing order of the global-access index of the first access in the pair (line 13). Then, the algorithm iterates over the ordered pairs in $orderedPairs$ (lines 15–25). The nested for loops iterate exactly n by k times, where n is the number of pairs in $orderedPairs$, and k is the window size. Note that k is a key parameter, as it controls both the accuracy and the overall complexity of our technique, it motivates our study (Study 1 in [68]), we empirically show that k is a relatively small size. Thus, for any pair $p1$ and $p2$, the algorithm checks for a pattern with $p1$ and $p2$ when they are within the sliding window.

Figure 6(b) shows the way in which the sliding window mechanism works for pairs. The left side of the figure (labeled Ordered pairs) gives the collected pairs ordered by the global-access index. The middle of the figure (labeled Pair window) shows how the sliding window with size 4 works. The sliding window has the first four patterns, shown within the dotted-line box, and the algorithm finds the

Table 5: Appearances and suspiciousness values of collected pairs (I1 to I4) and combined patterns (I5) in six executions of the program in Figure 4.

ID	Pairs/Patterns	E1	E2	E3	E4	E5	E6	Susp
I1	W_{1,S_1}(TABLE)W_{2,S_2}(TABLE)	√		√		√	√	0.5
I2	W_{2,S_3}(LOG)W_{1,S_4}(LOG)		√		√	√	√	0.5
I3	W_{2,S_2}(TABLE)W_{1,S_1}(TABLE)		√		√			0.0
I4	W_{1,S_4}(LOG)W_{2,S_3}(LOG)	√		√				0.0
I5	W_{1,S_1}(TABLE)W_{2,S_2}(TABLE) W_{2,S_3}(LOG)W_{1,S_4}(LOG)					√	√	1.0
		P	P	P	P	F	F	

$R_{1,S_1}(x)W_{2,S_2}(x)R_{1,S_3}(x)$ pattern within the window. Then, the window slides to the next four pairs, shown within the solid-line box, and the algorithm finds the $W_{4,S_4}(z)W_{2,S_5}(z)W_{2,S_6}(y)W_{4,S_7}(y)$ pattern within the window. The right side of the figure (labeled Patterns) shows the two patterns.

Pattern combination Given two pairs, Step 2 checks whether the pairs represent a pattern (P4-P17) (line 19). First, the algorithm checks whether the pairs belong to the same thread; if not, they cannot be combined. Then, the algorithm checks whether the pairs are in the form of P4 to P17; if so, the algorithm creates a pattern, and adds the pattern to *patterns* (lines 20–21).

4.2.3 Example

Table 5 shows how the algorithm works for Steps 1 and 2. The table shows appearances of collected pairs and combined patterns for six executions of the program in Figure 4. The first column shows the IDs. The second column shows the collected pairs and combined patterns. The third (E1) to eighth (E6) columns show appearances of the pairs and patterns in six program executions. The ninth column is the suspiciousness values, which will be explained in Section 4.2.4. I1–I4 are collected pairs with two memory accesses, and I5 is a combined pattern with four memory accesses. The final row shows program execution outcomes as passing (P) or failing (F). Now, consider each execution from E1 to E6. In E1, Step 1 detects and records two pairs, W_{1,S_1}(TABLE)W_{2,S_2}(TABLE) and W_{1,S_4}(LOG)W_{2,S_3}(LOG). Then, Step 2 tries to create a

new pattern consisting of the two pairs, so it checks whether the two pairs represent a new pattern. Step 2 finds that the two pairs are from the same threads, but the combination of the two pairs (i.e., W_{1,S_1}(TABLE)W_{2,S_2}(TABLE)W_{1,S_4}(LOG) W_{2,S_3}(LOG), or W_{1,S_4}(LOG)W_{2,S_3}(LOG)W_{1,S_1}(TABLE) W_{2,S_2}(TABLE)) are not listed as a pattern in Table 1. Thus, Step 2 cannot create new patterns from E1. In the same way, Step 2 cannot create any new pattern from E2 to E4. However, Step 2 can create a pattern I5, W_{1,S_1}(TABLE)W_{2,S_2}(TABLE) W_{2,S_3}(LOG)W_{1,S_4}(LOG), in E5 and E6.

4.2.4 Step 3: Rank Patterns

Step 3 of the algorithm computes a rank for each pattern, and presents the result to a developer. Step 3 inputs combined patterns *patterns* and program-execution outcome *outcomes*, and outputs the ordered list of patterns in decreasing order of suspiciousness as *rankedPatterns*.

The algorithm first associates patterns with program-execution outcome; *patterns-ToOutcome* records the number of occurrences of a pattern in passing and failing executions (lines 27–34). Then, the algorithm computes the suspiciousness of each pattern with its number of occurrences in passing and failing executions, and records the result in *suspmap* (lines 35–40). Finally, the algorithm ranks the patterns in decreasing order of suspiciousness, and returns the result (lines 41–42).

Our technique uses the statistical fault-localization technique, introduced by Jones, Stasko, and Harrold [39], to compute the suspiciousness of the patterns (line 39). This technique uses statistical analysis for fault localization for sequential or deterministic programs, and since it was introduced, there have been other techniques that provide different statistical formulas for the fault localization [6, 42, 43, 76] These approaches assume that entities (e.g., statements, branches, and predicates) executed more often by failing executions than passing executions are more suspicious of being the cause of the failure. Thus, they associate each entity with a suspiciousness score that reflects

this hypothesis.

For concurrent programs, we apply the same methodology to score suspiciousness of patterns. We found that the *Jaccard index* [6] best addresses the situation in which a pattern occurs in only one failing execution and no passing executions but is not related to an actual fault. For a pattern p, where $passed(p)$ is the number of passing executions in which we observe p, $failed(p)$ is the number of failing executions, and $totalfailed$ is the number of total failures, we use the following score:

$$\text{suspiciousness}_J(p) = \frac{\text{failed}(p)}{\text{totalfailed} + \text{passed}(p)} \tag{3}$$

Consider again the example in Table 5. The total number of failures is two. I1 appears in two failing and two passing executions. Thus, the suspiciousness of I1 is 0.5 using Formula (3). However, the suspiciousness of I5 is 1.0 because the pattern appears only in failing executions. In fact, I5 is the actual multi-variable atomicity violation.

4.3 Evaluation

To evaluate the UNICORN technique, we implemented prototypes for both C++ and Java, and used the prototypes to perform empirical studies with a number of C++ and Java subjects. Section 5.4.1 describes the implementation, and Section 4.3.2 describes the empirical setup. Then, Sections 4.3.3 presents the study. Finally, Section 4.3.4 discusses the threats to the validity of the studies.

4.3.1 Implementation

We implemented two modules for our technique (see Section 4.2): one module for Step 1, and the other module for Steps 2 and 3. For Step 1, we implemented modules in Java and C++. The modules takes a program written in Java or C++, and instrument shared-variable accesses in the program, so that the instrumented program will dynamically output memory-access pair information.

For Java, we used the Soot Analysis Framework,[2] which analyzes programs in Java bytecode format. UNICORN uses static escape analysis to determine possibly thread-escaping variables [24], and instruments read and write accesses of the shared variables in the program. UNICORN provides an option to inject artificial-delays that can increase the number of interleavings that occur, thereby increasing the chance of eliciting concurrency bugs [17, 89]. We use this option in our experiments. The UNICORN dynamic monitor executes in a separate thread as the instrumented program executes. This monitor dynamically receives memory-access information generated from multiple threads in a non-blocking concurrent queue, which maintains memory accesses in a sequential order. In this queue, we added a global access index counter, which is updated for every shared-memory access (recall Section 4.2.1). The accesses are obtained from this queue to construct windows for extracting pairs of accesses.

For C++, we created a module to instrument the subject programs statically using the LLVM analysis framework.[3] The module performs dynamic thread-escape analysis to find possibly shared variables in the program [90]. Then, the module instruments the escaping load and store instructions and memory operations, such as `malloc`, `memcpy`, and `memset`. During runtime of an execution of the instrumented program, the module uses the dynamic library-interposition method [32] to collect memory-access pairs.

The Java and C++ modules for Step 1 generate output files in XML format. These files contain memory-access pairs and the program execution outcome for each execution. For Steps 2 and 3, we implemented a module in Java that uses the XML files generated in Step 1. For Step 2, the module reads each XML file that has memory-access pair information and generates a new XML file that has memory-access pattern information. For Step 3, the module inputs the new XML files generated in Step 2,

[2] `http://www.sable.mcgill.ca/soot/`
[3] `http://www.llvm.org`

Table 6: Subjects used in evaluating UNICORN.

Type	Program	%Failed	PID	LOC	Type of Bug
C++ extracted	TimerThread	14.4	P2	68	Order
	LoadScript	49.8	P5	110	Single-variable atomicity
	MysqlLog	2.8	P6	89	Single-variable atomicity
	JsString	1.4	P12	95	Multi-variable atomicity
	MysqlDelete	3.8	P9	103	Multi-variable atomicity
	MysqlSlave	0.4	P14	94	Multi-variable atomicity
C++ complete	PBZip2	2.8	P2	2K	Order
	Mysql-791	64.0	P6	372K	Single-variable atomicity
	AGet	49.0	P9	1.2K	Multi-variable atomicity
	Mysql-169	63.0	P9	331K	Multi-variable atomicity
Java library	StringBuffer	22.3	P14	1.4K	Multi-variable atomicity
	Vector	7.6	P14	9.5K	Multi-variable atomicity

computes suspiciousness for each pattern, and generates the result.

4.3.2 Empirical Setup

Table 6 lists the subject programs we used for our studies [94]. The first column shows the types of the subject programs in three categories: C++ extracted programs, which are extracted buggy parts of program code from Mozilla and MySQL; C++ complete programs, which are complete applications without any simplification; and Java library programs, which are extracted classes from the Java 1.4 library. The second column shows the name of the subject program. The third column shows the failure rate that we observed empirically with our test cases. The fourth column shows the pattern ID from Table 1. The fifth and sixth columns list the size of the program in lines of code and the type of the concurrency bug, respectively.

We created test cases for the subjects. For the C++ extracted programs, we created test cases to call the extracted buggy parts concurrently and to determine program execution outcome. For C++ complete programs, we provided inputs to the subjects that can trigger concurrency bugs. `PBZip2` is an application that compresses files using the bzip2 algorithm with parallel threads. We call `PBZip2` to compress a large file with a number of threads. `MySQL` is a widely used open-source database

Table 7: Effectiveness of UNICORN.

Program	# Pairs	# Patterns	Sp (P1)	Sp (P2)	Sp (Pt)	R (Pt)	# R 1
TimerThread	6	7	1.0	-	1.0	1	3
LoadScript	3	4	0.86	1.0	1.0	1	2
MysqlLog	4	5	0.02	1.0	1.0	1	2
JsString	5	8	0.04	1.0	1.0	1	3
MysqlDelete	7	8	0.04	0.27	1.0	1	1
MysqlSlave	6	11	0.01	0.66	1.0	1	1
PBZip2	59	333	0.42	-	0.42	1	6
Mysql-791	1082	10936	1.0	0.64	1.0	1	4
AGet	37	94	0.51	0.50	0.70	1	1
Mysql-169	894	9051	0.63	0.63	1.0	1	7
StringBuffer	8	18	0.58	1.0	1.0	1	3
Vector	11	25	1.0	1.0	1.0	1	4

application. We concurrently call several queries that can trigger bugs in the database server. `AGet` is a multi-threaded download accelerator. We call `AGet` to download a large file from the Internet, and send a unix signal to stop the program and to trigger the concurrency bugs in the program. For Java library programs, we created a test case that create objects of the Java library classes, and execute the methods of the objects concurrently.

We ran our studies on a linux desktop with i5 2.8 GHz CPU and 8 GB of memory. We used gcc 4.4 and Java 1.5.

4.3.3 Effectiveness of Unicorn

The goal of this study is to investigate how well UNICORN ranks the pattern of the actual bug. To do this, we set the access window size to 5, and collected memory-access pairs in the C++ and Java subjects. To increase the probability of program failures, we inserted random artificial delays into the subject programs [17,89]. Then, we executed each subject program 100 times. We set the pair window size to 100.

Table 7 shows the results of the study. The third and fourth columns report the number of collected pairs and the combined patterns collected for the study. The fifth to seventh columns report suspiciousness values of the first memory-access pair,

the second memory-access pair, and the memory-access pattern of the actual bug. The eighth column reports the rank of the pattern that represents the actual bug. The ninth column reports the number of patterns that rank first. For example, for `TimerThread`, there are six reported pairs and seven reported patterns. The subject has an order violation of $W_{1,S_i}(x)R_{2,S_j}(x)$ pattern, so we report the suspiciousness of the pair in the fifth and seventh columns, but we do not report any value in the sixth column. There are three patterns that rank first in the report. We investigated the three patterns manually, and found that one of the three patterns is an order violation, and two other patterns are atomicity-violation patterns, which always appear with the actual order violation.

We make several observations about the results of Study 1 that are summarized in Table 7. First, the patterns that are ranked first are all actual bugs for all the subject programs. Thus, for these subjects, our technique is effective in finding non-deadlock concurrency bugs, and developers can focus on the top-ranked patterns to locate the actual bug. Second, some pairs that consist of the actual bug have low suspiciousness, but the pattern that manifests the actual bug has a high suspiciousness score. Thus, our pattern-combination technique is effective. For example, consider `MysqlSlave`. The suspiciousness values of the memory-access pairs are low, because they appear in both passing and failing executions. However, the two pairs appear together only in failing executions, and the suspiciousness of the pattern formed from these two memory-access pairs is 1.0. Third, UNICORN sometimes reports several bugs at the top rank (e.g., `Mysql-169` reports seven patterns with rank 1). From manual inspection, we found that all patterns ranked at the top are directly related to the actual bug.

4.3.4 Threats to Validity

There are several main threats to validity of the studies. Threats to internal validity include the empirical setup for our empirical evaluation, and in particular the test suite we used for the study. We did not measure the coverage or diversity of the interleavings of the test suite. However, to collect passing and failing executions with different thread schedules, we ran the program many times, which is similar to what would be done in practice. In addition, to increase the probability of the failures, we use an artificial delay injection technique.

Threats to external validity limit the extent to which our results will generalize to other kinds of concurrent programs. Whereas there are large benchmark suites [10] and benchmark generators [30] for sequential programs, there is no large and widely accepted bug benchmark suites for concurrent programs, and thus this threat is common to all prior work in this area. To mitigate this problem, our benchmark suite includes most of the benchmarks used in other work, and furthermore, includes both Java and C++ programs.

Threats to construct validity include the way developers find and fix bugs. The metric we use assumes that developers inspect the program with the memory access patterns in the rank order, and stop inspection when they reach the fault. Although this process may not be the real debugging situation, this approach to assessing effectiveness is used in many previous studies for assessing the effectiveness of statistical fault localization.

4.4 Related Work

There is much research in finding concurrency bugs. In this section, we discuss existing techniques that find concurrency bugs and compare UNICORN to them. Table 8

Table 8: Comparison to related techniques: The first column shows the type of techniques, and the second, third, and fourth columns represent whether the techniques can detect order, single-variable atomicity, and multi-variable atomicity violations.

Techniques	Order	S-Atom	M-Atom
S-Atom Violation Detector [51, 70, 84]		√	
M-Atom Violation Detector [25, 53, 59, 88, 95]		√	√
General Concurrency Bug Detector [34, 52, 54, 83]	△[4]	△	△
Unicorn	√	√	√

qualitatively compares UNICORN to the related work in terms of concurrency bug-detection ability.

Much prior research on concurrency bug detection has focused on bugs involving a single variable (the second row in Table 8). Some techniques focus on finding single-variable atomicity violations. AVIO [51] collects benign patterns for single-variable atomicity violations from passing executions, and finds and reports patterns that are not included in the benign patterns from failing executions. CTrigger [70] and PENELOPE [84] statically examine possible atomicity violations from an execution trace, and dynamically verify the violations. FALCON [67] is one of the first techniques that can find both order violations and single-variable atomicity violations. Like UNICORN, FALCON collects memory-access patterns dynamically and computes the patterns with suspiciousness scores with the same overhead as UNICORN. However, the bug-detection ability of all these techniques, including FALCON, is limited to concurrency bugs involving a single variable, whereas UNICORN finds both single-variable and multi-variable concurrency bugs.

There are several existing techniques for detecting multi-variable atomicity violations (the third row in Table 8). Some techniques require programmer annotations, either to specify the atomic regions that should be protected from interleavings [25,88]

[4] △ implies that the techniques neither distinguish the types of concurrency bugs nor show the details of the causes of each concurrency bug.

or to specify groups of memory accesses in which the memory accesses should be serialized [53]. More recent techniques [59, 95] automatically infer atomic regions in passing program executions, and find or avoid multi-variable atomicity violations by monitoring memory operations on atomic regions. Compared to these approaches, UNICORN has two advantages. First, UNICORN does not require any annotations. Second, UNICORN also handles order violation, whereas the other techniques are limited to only atomicity violations.

There are several existing techniques for detecting non-deadlock concurrency bugs without identifying the root cause of the bug (the fourth row in Table 8). CCI [34] monitors and samples shared memory locations and reports the likely buggy location using statistical methods. Bugaboo [52] collects communication graphs that contain a list of memory locations between threads and reports the graph with suspiciousness ranking. Recon [54] extends Bugaboo to reconstruct the buggy source and sink locations of two different threads from the communication graphs. Defuse [83] monitors the memory-access pairs between threads and report the most suspicious pairs as possible concurrency bugs. UNICORN differs from these techniques in several ways. First, these techniques report bugs without the root causes, so developers need to manually find the root cause to check whether the bug is an order violation or an atomicity violation. Second, these techniques may not report all important locations of the bug, so developers may not fully understand and fix the bug using the bug report. For example, multi-variable atomicity violations require two pairs of memory accesses, but techniques, such as Defuse [83], report only one pair of accesses. In contrast, UNICORN reports the detailed bug information with patterns that show the cause of the bug and the complete pairs of memory accesses.

4.5 Summary

This chapter presents UNICORN, the first unified technique that detects and ranks both single-variable and multi-variable non-deadlock concurrency bugs using patterns. UNICORN collects memory-access pairs dynamically, and combines memory-access patterns from memory-access pairs offline. The technique has manageable runtime overhead, comparable to other techniques, while extending detection ability from single-variable concurrency bugs to both single- and multi-variable concurrency bugs. Our empirical studies show that UNICORN is effective for a suite of C++ and Java subjects.

CHAPTER V

FAULT EXPLANATION FOR CONCURRENCY BUGS

5.1 Introduction

Existing fault-localization techniques for concurrent programs, including FALCON and UNICORN, locate likely concurrency bugs as interactions of memory accesses between multiple threads [34, 54, 67, 68], such as pairs of memory accesses [34, 60], memory-access graphs [54], or memory-access patterns [22, 67, 68]. However, the techniques have several limitations. First, the techniques report only memory accesses and lose context information, such as the call stacks of these accesses. Thus, developers must infer such information from the bug report to fully understand the concurrency bug. Second, the techniques do not automatically cluster memory accesses that are responsible for the same bug. Thus, a developer may need to manually cluster the buggy accesses to understand the bug. Finally, the techniques do not explicitly handle multiple concurrency bugs and may miss reporting concurrency bugs when multiple bugs exist. Thus, developers may need to identify multiple bugs manually from the bug report.

Semi-automatic fix techniques make patches for concurrency bugs [33,35,44]. The techniques input concurrency bugs reported from bug detectors [29, 97] and produce patches that enforce correct orderings using synchronization operations. However, these techniques also have limitations. First, the techniques are not completely automatic, and require a developer's additional help, such as selecting a fix strategy [35]. Second, the techniques use only simple strategies, such as lock-insertion, which is only 20% of all fix strategies [50]. Put another way, simple fixes may treat symptoms, but addressing the true cause of a bug may require more insight from the developer.

To address the limitations of existing techniques, we developed a new fault-comprehension technique, which we call GRIFFIN. GRIFFIN provides a way to explain concurrency bugs using additional information over memory accesses, and thus, bridges the gap between fault-localization and fault-fixing techniques. GRIFFIN aids developers in understanding concurrency bugs by providing more information over fault-localization techniques, such as suspicious methods with which developers can easily locate atomic regions for atomicity violations. To assist automatic fault-fixing techniques, GRIFFIN again provides more information, such as clusters of memory-access patterns, that can be used to generate only one patch for each cluster.

The key idea in GRIFFIN is to group suspicious memory-access patterns. The process consists of three steps. In Step 1, GRIFFIN executes an existing fault-localization technique to obtain suspicious memory-access patterns that represent concurrency bugs and a coverage matrix that shows the occurrences of the patterns in testing executions. In Step 2, GRIFFIN uses the output of Step 1 to cluster executions that fail for the same concurrency bug. Finally, in Step 3, GRIFFIN clusters patterns in the grouped failing executions (obtained in Step 2) and reports information for each likely bug in the program: suspicious memory-access patterns, suspicious methods from which suspicious memories are accessed, and bug graphs that show the interactions of groups of memory accesses.

There are three main advantages of GRIFFIN over existing fault-localization techniques. First, GRIFFIN *assists developers in understanding bugs* by providing summarized information for each bug. A developer begins to understand the bugs at a high level using a bug graph that shows context changes on groups of memory accesses. Next, the developer may investigate suspicious methods that are likely to contain the locations to be fixed. Finally, the developer may investigate each access pattern to understand each raw-memory access. Second, GRIFFIN *handles multiple concurrency bugs* by grouping memory-access patterns. Thus, developers can concentrate on

investigating patterns in a cluster, instead of investigating an unclustered list of patterns. Moreover, the identified different bugs might be assigned to several developers and handled concurrently [37]. Finally, GRIFFIN *assists automatic fix techniques* by providing groups of fault-localization results. Thus, fix techniques can generate only one patch for each cluster instead of generating patches for each pattern and merging them for each bug afterward.

To evaluate GRIFFIN, we implemented it for both Java and C++, and performed empirical studies on a set of real programs. The first study investigates how well GRIFFIN clusters failing executions caused by the same bug. The results show that the technique accurately clusters failing executions with the parameters we used. The second study investigates how well GRIFFIN clusters the suspicious patterns associated with the same bug, and presents the details of the bug. The results show that our technique presents suspicious patterns, suspicious methods, and bug graphs with only a few false positives.

The main contributions of the work are summarized as follows:

- It identifies problems with existing fault-localization technique for concurrent programs.
- It presents the GRIFFIN technique, which clusters executions and patterns that are associated with the same bug, thereby providing a way to explain bugs.
- It carries out empirical studies that show the effectiveness of the GRIFFIN technique for clustering and explaining concurrency bugs.

5.2 Problems in Existing Techniques

This section presents an example bug (Section 5.2.1), discusses three common problems with the existing techniques (Section 5.2.2), and discusses the challenges related to addressing these problems (Section 5.2.3).

Thread 1

```
149  Vector(Collection c) {
150    size = c.size();
151    array = new Object[size];
152    c.toArray(array);
153  }

270  int size() {
271    return size; // read of this.
272  }

680  Object[] toArray(Object a[]) {
681    // read of this.size
682    // read of this.data
683  }
```

Thread 2

```
850  boolean addAll(Collection c) {
851    // write of this.size
852    // write of this.data
853  }

621  void removeAllElements() {
622    // write of this.size
623    // write of this.data
624  }

800  void remove(int index) {
801    // write of this.size
802    // write of this.data
803  }
```

Figure 8: Atomicity Violation in `Vector`: Atomicity violations can occur when the execution of the atomic region in the `Vector` constructor (lines 150–152) is interfered by any of the three methods in Thread 2. For example, an atomicity violation can occur between `Vector` and `addAll` with the following order: 150→151→851→852→152.

5.2.1 Example

We will use this atomicity violation example for this section. Figure 8 shows snippets from the `Vector` class of the Java Collection Library, which has an atomicity violation; statements in the code are labeled with their line numbers. Thread 1 executes the code on the left, and Thread 2 executes one of the three methods on the right. The constructor is the atomic region, and it must be executed without interference: `c` in lines 150–152 must be accessed without interference from the code in Thread 2. Otherwise, an atomicity violation can be triggered.

To illustrate, suppose the lines execute in order 150→271→151→851→852→152→681→682. In lines 150–151, the program retrieves `size` of `c` and initializes `array` of `Vector` with the `size`. However, in lines 851–852, the `size` and `array` of `c` are increased by the `addAll` method. Then, in lines 152–682, the increased `array` of `c` is copied to `array` of `Vector`, and thus, the program may crash with an exception. In this example, two patterns are manifested as atomicity violations. The first pattern involves a single variable, `size`: a read-write-read (RWR) pattern (i.e., the read accesses are from Thread 1, and the write access is from Thread 2), and the access

order is 271→851→681. The second pattern involves two variables, `size` and `array`: a read-write-write-read (RWWR) pattern (i.e., the read accesses are from Thread 1, and the write accesses are from Thread 2), and the access order is 271→851→852→682.

5.2.2 Problems

Existing fault-localization techniques share three common problems. Since UNICORN provides the most comprehensive reports, we illustrate these problems using UNICORN's reports, a sample of which appears in Table 9.

Table 9 shows the execution statistics of memory-access patterns and outputs reported by UNICORN for the example in Figure 8. The first column shows the pattern index (PI). The second to fourth columns are executions statistics. The second column reports the associated BugID (B), from 1 to 3, because there are three atomicity violations in the example. The third and fourth columns report the number of occurrences of the pattern in passing (P) and failing (F) executions, respectively. The fifth to ninth columns are outputs of UNICORN. The fifth and sixth columns show the suspiciousness score of the pattern (S) and the ranking of the pattern (R), respectively. The seventh column reports the type of the pattern, with memory access types. The final column reports the access locations of the pattern in the source code.

Problem 1 (Loss of Context): The first problem is that existing techniques report memory-access locations but not the context of the bug. UNICORN outputs *all* raw memory-access locations that are likely associated with concurrency bugs, but even all raw memory-access locations may not fully explain the cause of the bug. To illustrate, consider again the example in Figure 8. To understand the bug, the developer would inspect the code using one of the four first-ranked patterns in Table 9.[1] Then, the developer would locate the `size` and `toArray` methods, but may not easily find the

[1]Like many fault-localization techniques, if patterns result in the same suspiciousness, UNICORN gives them the same rank, and thus, they would be inspected in any order.

Table 9: Execution statistics for memory-access patterns and UNICORN output for the bugs in Figure 8.

PI	Statistics			UNICORN output			
	B	P	F	S	R	Pattern	Pattern accesses
1	1	0	3	0.60	1	RWR	271→851→681
2	1	0	3	0.60	1	RWWR	271→851→852→682
3	1	0	3	0.60	1	WR	851→681
4	1	0	3	0.60	1	RW	271→851
5	1	3	3	0.38	5	RW	852→682
6	2	0	2	0.20	6	RWR	271→622→681
7	3	0	2	0.20	6	RWR	271→801→681
8	2	0	2	0.20	6	RWWR	271→622→623→682
9	3	0	2	0.20	6	RWWR	271→801→802→682

constructor using these accesses. Figure 9 shows the same memory accesses of `size` along with the call stack for each memory access. Solid boxes represent methods on the call stack, and dashed boxes represent memory accesses. Without inspecting the figure, developers may not find that `Vector` is the common method that leads to the two raw-memory accesses.

Problem 2 (True-/False-positive Patterns): The second problem is that existing techniques report the ranked list of the patterns, but do not group patterns responsible for the bug. Consider again the example in Figure 8. As discussed in Section 2.1, two patterns can manifest the same atomicity violation. PIs 1 and 2 are atomicity violation patterns involving a single variable (i.e., `size`) and multiple variables (i.e., `size` and `array`), respectively. PIs 3–5 manifest the same violation, and the pattern accesses are part of PIs 1 and 2. Thus, PIs 1–5 manifest as one bug: B1. However, neither UNICORN nor the other techniques [34, 54, 83] reveal that these patterns are associated with the same bug.

Problem 3 (Multiple Bugs): The third problem deals with multiple bugs. PI6 to PI9 in Table 9 illustrate the problem. The PIs are associated with two different bugs. PI6 and PI8 are associated with B2, and PI7 and PI9 are associated with B3. B2 is

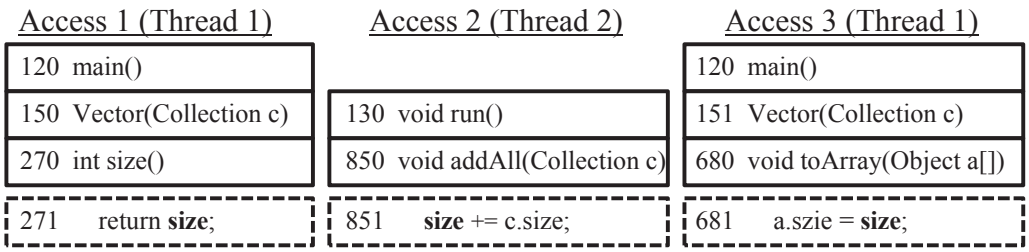


Figure 9: Problematic memory accesses with call stacks.

an atomicity violation that occurs between the `Vector` and the `removeAllElements` methods. B2 occurs when an execution order is 150→`removeAllElements`→152. Similarly, B3 is an atomicity violation that occurs between the `Vector` and the `remove` methods. B3 occurs with an execution order, 150→`remove`→ 152. When multiple bugs exist, the patterns manifesting different bugs can be listed with the same or similar suspiciousness score, and developers may need to associate the patterns with different bugs manually.

5.2.3 Challenges

The problems of Section 5.2.2 raise a number of challenges.

Challenge 1 (Efficient Information Gathering): Existing techniques collect and report only pairs of memory accesses or patterns of accesses (in UNICORN). This property helps to keep the amount of information gathered small, making the techniques efficient overall. However, to present concurrency bugs with high-level context and clusters of accesses, a new technique needs to collect more context information with accesses efficiently and process them accordingly.

Challenge 2 (Large Context): The second challenge is related to large size of calling contexts. It may be argued that Problem 1 can be easily addressed by reporting only a call stack for each memory access. However, the problem is not so easy. The stack sizes sometimes grow very large, such as `mysql-169`, where the size

of the call stacks grow to over 10 methods spread across five source files. In addition, the stack comparison involves not only stacks of two memory accesses, but stacks of patterns of memory-accesses, and moreover, the number of stacks grows to several thousands.

Challenge 3 (Large Number of Patterns): The third challenge is related to the large number of patterns. It may be argued that Problems 2 and 3 can be easily addressed by manual clustering of patterns by a developer. However, it may be difficult to group patterns manually when the size of the report grows. Our experience suggests that the size of a report, like `mysql-791` in Table 10, can grow to more than 10K in some subjects [68], and the size of the patterns responsible for the same bug may grow accordingly.

5.3 Technique

Griffin consists of three steps as shown in Figure 10. Step 1 takes as input a concurrent program P and a test suite T. Griffin executes P with T multiple times, collecting memory-access patterns from each execution and labeling executions as passing or failing. Then, Griffin computes a ranked list of patterns, F, using a fault-localization scheme, and generates a coverage matrix, M, that represents the associations of the patterns and the executions. Step 2 uses F and M to cluster executions that likely failed due to the same bug. This clustering is based on Algorithm 1, ClusterFailingExecutions (Algorithm 1), and underlies Griffin's ability to handle multiple bugs. The output of Step 2 is a set of clusters, C, where each cluster contains a set of failing executions. Finally, Step 3 executes algorithm ClusterSuspiciousPatterns (Algorithm 2) on F, M, and C. This algorithm groups patterns for constructing higher-level context that will be of direct use to the developer in identifying the bug. The output C' is a set of clusters, where each cluster contains a set of patterns. For each cluster in C', Step 3 also reports two suspicious methods

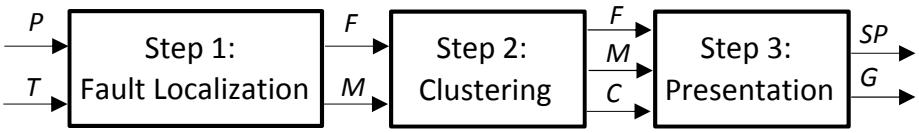


Figure 10: Overview of GRIFFIN.

(one for each thread in the cluster), $Meth$, that are likely to contain the location to be fixed, and a bug graph G, that visually shows the bug.

5.3.1 Step 1: Localize Problematic Patterns

Step 1 is based on the UNICORN [68] fault-localization technique. We extended UNICORN in two ways to provide F and M. First, we extended UNICORN to record call-stack information for each shared-memory access, and thus, accesses in the patterns in F are also associated with call stacks. This modification is straightforward and is used in other techniques [33, 35, 44].

Second, we extended UNICORN to report M. UNICORN maintains M internally in its pattern-combination step, so we changed UNICORN to output the information. Figure 11 shows an example of M, for the bug example in Figure 8 and Table 9. In the figure, the rows represent the pattern indexes (PI1–PI15); the first 15 columns represent the executions (E1–E15); the labels at the bottom of these columns represent the results of those executions (P for passing and F for failing); and the last two columns represent the suspiciousness score and the rank of the PIs, respectively. A solid circle in a cell in M, denoted by (PI,E), indicates that PI is observed in the execution of E. For example, the solid circle in (PI1, E1) indicates that pattern accesses of PI1 were observed in the execution E1, the suspiciousness score of PI1 is 0.6, and PI1 is ranked 1st. Note that Figure 11 has nine PIs in common (i.e., PI1–PI9) with Table 9, but it has six more PIs (i.e., PI10–PI15) for illustration of our technique.

	E1	E2	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	Susp	Rank
PI1	●	●	●													0.60	1
PI2	●	●	●													0.60	1
PI3	●	●	●													0.60	1
PI4	●	●	●													0.60	1
PI5	●	●	●					●	●			●				0.38	5
PI6				●	●											0.20	6
PI7						●	●									0.20	6
PI8						●	●									0.20	6
PI9				●	●											0.20	6
PI10				●	●			●	●			●	●	●	●	0.15	10
PI11						●	●		●	●	●	●	●	●		0.15	10
PI12				●		●		●	●	●	●	●	●		●	0.14	12
PI13					●		●	●	●		●	●	●	●	●	0.14	12
PI14				●			●	●	●	●	●	●	●	●	●	0.13	14
PI15	●							●	●	●	●	●	●	●	●	0.06	15
	F	F	F	F	F	F	F	P	P	P	P	P	P	P	P		

Figure 11: Coverage matrix for the atomicity violations in Figure 8 and Table 9.

5.3.2 Step 2: Cluster Failing Executions

The main purpose of Step 2 is to handle multiple bugs by grouping executions that fail for the same reason. To do this, Step 2 executes a clustering algorithm based on the results of Step 1, and outputs a set of clusters of failing executions for use in Step 3.

There are several metrics to use for clustering algorithms, such as statement, branch, definition-use profiles, memory-access pairs between threads, and memory-access patterns. We tried different metrics and used the fault-localization results (i.e., ranked memory-access patterns) to develop our clustering algorithm for two reasons. First, fault-localization results are available from Step 1, so Step 2 can use them without collecting additional profile data. Second, concurrency bugs usually occur in specific thread interleavings and are related to a small portion of program profiles,

but are not highly related to the entire profile of the program execution.

Figure 11 illustrates the intuition behind our approach. Executions E1–E7 are failing executions: E1–E3 fail because of the same bug; E4 and E5 fail because of the same bug; and E6 and E7 fail because of the same bug. Each failing execution associated with the same bug has a similar set of top ranked patterns. Consider E1–E3. PI1–PI4 are highly ranked and appear only in these executions. In contrast, PI10–PI15 are lowly ranked, and mainly appear in passing executions. Even if the pattern accesses appear in failing executions, they appear randomly. Therefore, using the top ranked patterns for each failing execution will facilitate the clustering of executions that fail for the same bug.

Algorithm 3, our fault-localization-based clustering algorithm, inputs a ranked list of memory-access patterns, F, a coverage matrix, M, the number of suspicious patterns to be considered, t, and the threshold of similarity of two clusters, s, and outputs a set of clusters, C=$\{C_1, C_2, \ldots, C_k\}$, where each cluster C_i is a set of failing executions. The algorithm starts by initializing each cluster to contain one failing execution (lines 1–4). Then, the algorithm iterates until there are no more merged clusters (lines 5–27). In this main loop, the algorithm compares every pair of clusters to determine whether they can be merged (lines 10–26). To check the similarity, the algorithm fetches the top t patterns in two clusters (lines 11–12), and computes the similarity of the two clusters (line 13). For the similarity computation, the algorithm uses the Jaccard similarity index, which is shown in Equation 4. The value of the index ranges from 0 (completely different) to 1 (completely alike).

$$\text{Similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{4}$$

The algorithm keeps the maximum similarity by comparing the similarity of the current pair and *max_similarity* (lines 14–17). If the two clusters are exactly the same (lines 18–20), the algorithm simply merges the two clusters (lines 22–26). Otherwise, if the maximum similarity is greater than the threshold (lines 22–26), the algorithm

Algorithm 3: ClusterFailingExecutions

Input : F: ranked list of memory-access patterns
M: coverage matrix
t: number of suspicious patterns to be considered
s: threshold of similarity of two clusters
Output: C: set of clusters, each of which is a set of failing executions
Data: k: number of clusters
merged: true if any two clusters were merged

```
1  Set k = number of failing executions
2  for failing executions, f_i, where 1 ≤ i ≤ k do
3  |  C_i = {f_i}
4  end
5  merged = true
6  while merged do
7  |  merged = false
8  |  max_similarity = 0
9  |  cand_pair = ∅
10 |  for all (C_i, C_j) where i≠j and 0≤i,j≤k do
11 |  |  R_i = getSuspiciousPatterns(C_i, M, F, t)
12 |  |  R_j = getSuspiciousPatterns(C_j, M, F, t)
13 |  |  similarity = getSimilarity(R_i, R_j)
14 |  |  if similarity > max_similarity then
15 |  |  |  cand_pair = (C_i, C_j)
16 |  |  |  max_similarity = similarity
17 |  |  end
18 |  |  if max_similarity == 1 then
19 |  |  |  break
20 |  |  end
21 |  end
22 |  if max_similarity ≥ s then
23 |  |  merge cand_pair
24 |  |  k = k - 1
25 |  |  merged = true
26 |  end
27 end
28 C = {C_1, C_2, ..., C_k}
29 return C
```

merges the two clusters. The algorithm terminates the main loop when no more clusters are merged, and returns the clusters (line 28).

Consider again Figure 11. Algorithm ClusterFailingExecutions starts by initializing each cluster to one failing execution: for $i \in 1 \ldots 7$, C_i={Ei}. Suppose

t=5 and s=0.5. Consider C_1 and C_2 as the first pair of clusters. These clusters can be merged because the five top-ranked patterns of each cluster are exactly the same and thus, the similarity is 1 (lines 18–20 and 22–26). Let C'_1={E1, E2}. Note that the algorithm breaks out of the comparison loop if it finds two identical clusters (line 15). Now consider C'_1 and C_3 as the next pair of clusters. This pair can also be merged because the five top-ranked patterns of each cluster are exactly the same resulting in a similarity of 1. Let C''_1={E1, E2, E3}. Another candidate pair is C_4 and C_5. E4 has five patterns, and E5 has four patterns. Because three patterns appear in both E4 and E5 and six distinct patterns appear in E4 and E5, the similarity is $\frac{3}{6}$ or 0.5. Thus, the pair can be merged. Likewise, E6 and E7 can be merged. Finally, the algorithm reports three clusters: C''_1={E1, E2, E3}, C'_2={E4, E5}, and C'_3={E6, E7}.

5.3.3 Step 3: Reconstruct Bug Context

The main purpose of Step 3 is to reconstruct high-level bug context from the clustered patterns to assist understanding of the bug. To do so, Step 3 fetches the top-ranked patterns for each group of failing executions reported from Step 2, and distinguishes the true and false positive patterns with respect to the bug. Step 3 outputs a set of clusters of patterns, C', for each cluster in C. Then, for each cluster of patterns in C', Step 3 merges accesses of the patterns, and reports suspicious methods, $Meth$, and a bug graph, G.

We use a clustering algorithm based on the similarity of the call stacks [13]. The intuition behind the call-stack-similarity-based clustering is that accesses appearing closely in execution are likely to have similar call stacks, and thus, accesses responsible for the same concurrency bug are likely to have similar call stacks. The same intuition is applied to patterns. If two patterns are responsible for the same bug, the call stacks in the patterns are likely to be similar. Specifically, we use common call stacks to

Algorithm 4: ClusterSuspiciousPatterns

Input : F: ranked list of memory-access patterns
M: coverage matrix
C_f: a set of failing executions
t: number of suspicious patterns to be considered
Output: C': set of clusters, each of which is a set of patterns
Data: k: number of clusters

```
1  Set k = t
2  C' = getSuspiciousPatterns(C_f, M, F, t)
3  for patterns p_i in C' do
4  |  CL_i = {p_i}
5  end
6  merged = true
7  while merged do
8  |  mergeable = false
9  |  for all (CL_i, CL_j) where i≠j and 0≤i, j≤k do
10 |  |  mergeable = isMergeable(CL_i, CL_j)
11 |  |  if mergeable then
12 |  |  |  merge(CL_i, CL_j)
13 |  |  |  k = k - 1
14 |  |  |  merged = true
15 |  |  |  break
16 |  |  end
17 |  end
18 end
19 C' = {CL_1, CL_2, ..., CL_k}
20 return C'
```

compare patterns. A *common call stack* of a group of memory accesses is the common part of the call stacks from the bottom of the stacks. Consider accesses 1 and 3 in Figure 9. The common call stack of the two accesses is the common part of the two call stacks of the accesses: `main()` and `Vector()`.

Algorithm 4, our suspicious-pattern clustering algorithm, inputs a ranked list of memory-access patterns F, a coverage matrix M, the number of suspicious patterns to be considered t, and a set of failing executions, C_f, that fail because of the same bug. The algorithm starts by fetching the top t patterns from C_f and initializing each cluster to a pattern (lines 1–4). Then, the algorithm iterates until there are no more merged clusters (lines 7–17). In the main loop, the algorithm compares every pair of

clusters to determine whether the pair can be merged (lines 9–15). The algorithm terminates the main loop when no more clusters are merged, and returns the clusters (line 19).

The main part of the algorithm is the `isMergeable` function that determines whether two clusters of patterns can be merged. Consider two clusters, CL_i and CL_j. There are two cases. First, if CL_i and CL_j have common call stacks that are the same, the clusters can be merged. Second, if CL_i has only one pair and the pair in CL_i is part of the triple or quadruple in CL_j, then the clusters can be merged.

Consider again Table 9 and Figures 9 and 11 that were used in Step 2. Step 3 inputs C_1''={E1, E2, E3}, C_2'={E4, E5}, and C_3'={E6, E7}. Suppose t=5, and we run Algorithm 4 for C_1''. The top five patterns in C_1'' are PI1–PI5. The algorithm starts by initializing each cluster with each pattern: for $i \in 1 \ldots 5$, CL_i={PIi}. First, the algorithm compares CL_1 and CL_3. Each cluster has one pattern, and PI3 (i.e., 851→681) is part of PI1 (i.e., 271→851→681). Thus, they are merged by exact subsumption, and let CL_1'={PI1, PI3}. Similarly, PI4 and PI5 are merged to PI2, and let CL_2'={PI2, PI4, PI5}. Now, the algorithm compares CL_1' and CL_2', and merges them by call stack similarity. The common call stacks of the clusters are: `main()` and `Vector()` from one thread; `run()` and `addAll()` from the other thread as in Figure 12. Finally, the algorithm reports only one cluster, CL_1''={PI1, PI2, PI3, PI4, PI5}, for C_1''.

Step 3 reports two suspicious methods $Meth$ one for each thread in the cluster. We define a *suspicious method* as the method at the top in the common call stack. A cluster has two groups of memory accesses for each thread, and thus, Step 3 finds two suspicious methods, one for each thread. Consider again Figure 9. Suppose a cluster has only these three accesses. Then, a group of memory accesses for Thread 1 is Access 1 and Access 3, the other group of memory accesses for Thread 2 is Access 2, and the suspicious methods are `Vector()` and `addAll()`.

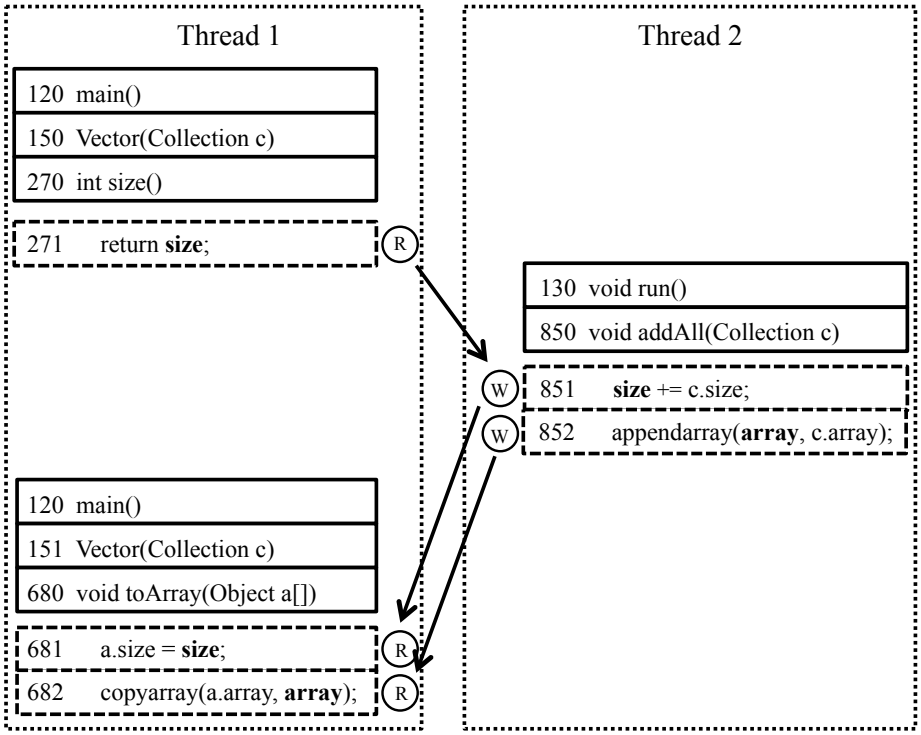


Figure 12: Bug graph for the atomicity violation B1 in Table 9.

Step 3 reports a *bug graph* G for each cluster of patterns. To create the graph, Step 3 divides memory accesses into two groups, one for each thread. Then, Step 3 investigates memory-access orderings and groups accesses that have the same ordering sequences. Figure 12 shows an example of a bug graph for the cluster, CL''_1. Each node has a group of memory accesses with a common call stack. Solid rectangles represent methods on the call stack, and dotted rectangles represent memory accesses. For example, the node in Thread 2 has two write accesses, lines 851 and 852, and their common call stack has two methods. Edges between threads show the orderings of the memory accesses, and in this example, the edges can represent all five patterns in PI1–PI5. For example, PI1 is a RWR pattern of 271→851→681 accesses, and we can find the corresponding edges of the arrows in the graph.

5.4 Evaluation

To evaluate the GRIFFIN technique, we implemented it in a prototype tool, and evaluated the tool on a set of subjects.[2] This section describes the implementation (Section 5.4.1) and the empirical set up (Section 5.4.2), presents our studies (Sections 5.4.3–5.4.4), and discusses the threats to the validity of the studies (Section 5.4.5).

5.4.1 Implementation

We implemented our prototype tool for Java and C++. For Step 1 of our technique, we implemented modules for both Java and C++. The modules are based on the implementations of UNICORN [68], and we extended them to record call stacks for each shared-variable access and to provide the coverage matrix. For both languages, Step 1 outputs the results in XML format. We implemented the module using the Soot framework[3] and the PIN binary instrumentation tool[4] for Java and C++, respectively.

For Step 2, we implemented the module in Java. The module takes XML files containing the results of Step 1 as input, and produces as ouput a text file of clusters of failing executions. Finally, for Step 3, we implemented the module in Python and GraphViz. The module reads the XML and the text files, and produces clusters of patterns and suspicious methods in text files, and bug graphs in PNG graphics files.

5.4.2 Empirical Setup

Table 10 lists the subject programs we used for our studies. The first column shows the subject's language (Language), either Java or C++. The second column shows the name of the subject program (Program). The third column shows the failure rate that we observed empirically with our test cases (% Fail). The fourth column shows the number of concurrency bugs that we identified with our test cases (# Bugs). The

[2]The details of our empirical studies are available at `http://www.cc.gatech.edu/~sangminp/griffin/`

[3]`http://www.sable.mcgill.ca/soot/`

[4]`http://pintool.org/`

Table 10: Subjects used in evaluating Griffin.

Language	Program	% Fail	# Bugs	KLOC	Type
Java	TreeSet-1	42.0	5	7.5	Atomicity
	TreeSet-2	29.0	3	7.5	Atomicity
	StringBuffer-1	33.0	4	1.4	Atomicity
	StringBuffer-2	18.0	1	1.4	Atomicity
	Vector-1	8.0	4	9.5	Atomicity
	Vector-2	14.0	2	9.5	Atomicity
C++	Mysql-169	29.0	1	331	Atomicity
	Mysql-791	24.0	1	372	Atomicity
	NSPR-165586	18.0	1	125	Atomicity
	PBZip2	75.0	1	2	Order
	Transmission	31.0	1	90	Order

fifth column lists the size of the program in KLOCs. Finally, the sixth column shows the type of the concurrency bug (Type), either an atomicity or an order violation. For example, `TreeSet-1` is a Java program that fails 42% of the time with our test suite. The size of the program is about 7,500 lines of code, and it fails with five different atomicity violations.

We created test cases, and ran them multiple times for each subject to collect various interleavings from multiple executions. For the Java programs, which are a set of classes in the Java collection library, we created two test cases for each program. These test cases execute different parts of code with different levels of concurrency, and thus, programs with different test cases may exhibit different sets of bugs. Specifically, the program name with suffix 1 has a test case that creates four threads and calls two methods for each thread, and the program name with suffix 2 has a test case that creates two threads and calls one method for each thread. `Mysql-169` and `Mysql-791` are two versions of `Mysql` open-source database, and `NSPR-165586` is a library of the Mozilla open-source browser. For each program, we created test cases as described in the program's bug repository to emulate the real debugging environment of the developers. `PBZip2` is a compression utility program, and we provided a large file for

the program to compress. `Transmission` is a torrent download utility, and we used a torrent file for the program to download and observed whether the program fails.

We ran our studies on a Linux desktop with 2.8 GHz CPU and 8 GB of memory. We used gcc 4.1 and Java 1.5.

5.4.3 Study 1: Handling Multiple Bugs

The goal of this study is to investigate how well GRIFFIN handles multiple concurrency bugs by examining the clusters of failing executions it produces. More specifically, we collected the fault-localization results from Step 1 of our technique, and ran Step 2 on the results. Recall that Algorithm 3 in Step 2 has two parameters, t and s: t is the number of suspicious patterns to be considered for each failing execution, and s is the threshold of similarity of two clusters. We performed a preliminary study that investigates a range of values of t and s to determine the parameter values that are likely to report the same number of clusters as the number of bugs. From this study, we let $t = 30$ and $s = 0.8$.

To evaluate the effectiveness of the clustering, we used the F-measure metric. This metric is based on a combination of precision and recall as used in the information-retrieval domain, and is widely used to evaluate clustering algorithms [13, 85]. In this case, *precision* refers to the proportion of clustered failing executions that are relevant, *recall* refers to the proportion of relevant failing executions that are clustered; the *F-measure* is a weighted combination of precision and recall.

We denote C as the set of clusters. Specifically, let C_i be the ith cluster that our algorithm reports, O_j be the jth optimal cluster, and N be the total number of failing executions. Then, we calculate precision and recall as follows.

$$\text{Precision}(C_i, O_j) = \frac{|C_i \cap O_j|}{|C_i|},\ \text{Recall}(C_i, O_j) = \frac{|C_i \cap O_j|}{|O_j|}$$

We use Van Rijisbergen's F-measure [85], which computes the weighted average of

Table 11: Study 1: Handling multiple bugs.

Program	# Patterns	# Optimal	# Clusters	F-measure
TreeSet-1	714	5	7	0.88
TreeSet-2	656	3	4	0.91
StringBuffer-1	12	4	4	1.00
StringBuffer-2	3	1	1	1.00
Vector-1	18	4	4	1.00
Vector-2	10	2	2	1.00
Mysql-169	21834	1	1	1.00
Mysql-791	71694	1	2	0.94
NSPR-165586	1479	1	2	0.86
PBZip2	427	1	2	0.96
Transmission	226	1	1	1.00

maximal F-measure values for each cluster. The equations are as follows.

$$F(C_i, O_j) = \frac{2 * \text{Recall}(C_i, O_j) * \text{Precision}(C_i, O_j)}{\text{Recall}(C_i, O_j) + \text{Precision}(C_i, O_j)}$$

$$F\text{-measure}(C) = \sum_i \frac{|C_i|}{N} * \max_j\{F(C_i, O_j)\}$$

The F-measure value ranges from 0 (least effective) to 1 (most effective).

Table 11 shows the results of the study. The first column shows the program name (Program). The second column shows the number of patterns (# Patterns) that the fault-localization technique in Step 1 reports. The third column shows the optimal number of failing execution clusters (# Optimal), which should be the same as the number of bugs in the program. The fourth column shows the number of clusters that Griffin reports (# Clusters). Finally, the fifth column shows the F-measure value (F-measure). For example, for `Mysql-791`, Griffin reports 71,694 patterns after Step 1. The optimal number of clusters is one, and Step 2 of the technique reports two clusters. The F-measure value is 0.94.

The main observation from Table 11 is that most of the F-measure values are 1.00 or are close to 1.00, and none is less than 0.86. The high F-measure values indicate that the failure clustering algorithm is effective in handling multiple concurrency

bugs with our parameter settings. One important parameter is t, which we set to 30. Although previous techniques [33, 35, 54, 67, 68] reveal that several patterns may indicate the same bug, they do not report the number of patterns that are related to the same bug. Our study showed that, for our subjects, up to 30 patterns are related to the same bug.

We performed a detailed investigation of the subjects whose F-measure values are not 1.00. For example, for `Mysql-791`, we used 24 failing executions as input to the clustering algorithm. The optimal number of clusters is one, but the algorithm reported two clusters, one with 23 failing executions and the other with one failing execution. We manually investigated these clusters, and found that these two clusters are also similar. We also found that, when we used a lesser value for s (the threshold of similarity), we get the optimal number of clusters for the subject. Thus, we believe that there may be a better combination of parameters that more often gives optimal results, and we plan to investigate this in future work.

Another observation involves the "super-bug effect" [98]: for sequential programs, the clustering techniques often did not cluster failing test cases effectively when the program has multiple bugs, and one bug hides the appearance of the other bug. For concurrent programs and test cases we investigated, we found no executions that reveals a super bug.

5.4.4 Study 2: Reconstructing Bug Context

The goal of this study is to investigate how well Griffin reconstructs bug contexts. Specifically, this study investigates how well Griffin clusters true positive patterns and locates concurrency bugs in both method and access levels. To do so, we run Step 3 of the technique on the optimal clusters of failing executions that Step 2 produces. Recall that Algorithm 4 in Step 3 has a parameter t, which is the number of patterns to be re-clustered. We set $t = 20$ based on the observation that only top-ranked

Table 12: Study 2: Reconstructing bug context.

Program	# Optimal	# Clusters	FP	Meth	Call Size
TreeSet-1	5	5	0	Y	6
TreeSet-2	3	3	0	Y	6
StringBuffer-1	4	4	0	Y	1
StringBuffer-2	1	1	0	Y	1
Vector-1	4	4	0	Y	1
Vector-2	2	2	0	Y	1
Mysql-169	1	2	1	Y	9
Mysql-791	1	1	0	Y	1
NSPR-165586	1	1	0	Y	4
PBZip2	1	1	0	Y	0
Transmission	1	1	0	Y	7

patterns are likely to be responsible for the real bugs.

To evaluate the effectiveness of the technique, we investigated the outputs of Step 3: clustered patterns, suspicious methods, and bug graphs, using our understanding of the bugs. To evaluate the effectiveness of the clustering, we checked whether the bug graph of the cluster has orderings and contexts consisting of the bug. If the bug graph does not represent a bug, we set it as a false positive. To evaluate the effectiveness of the bug locating ability, we manually checked whether the suspiciousness method locates the method containing the cause of the bug. For example, if the bug is an atomicity violation, we checked whether the suspicious method locates the atomic region; if the bug is an order violation, we checked whether the suspicious method has the location in which incorrect orderings occur.

Table 12 shows the results of the study. The first column shows the program name (Program). The second column shows the number of optimal clusters of patterns (# Optimal), which is the same as the number of bugs. The third column shows the number of clusters that Griffin reports (# Clusters), The fourth column shows the number of false-positive clusters (FP), which is the difference between the values in the third and second columns. The fifth column shows whether the suspicious

methods locate the cause of the bug (Meth). Finally, the sixth column shows the maximum size of call stacks between suspicious methods and raw-memory accesses (Call Size). For example, GRIFFIN reports two clusters for `Mysql-169`, where the patterns in one cluster indicate the real bug, and the other cluster is a false positive. The suspicious methods locate the atomic region, and the maximum size of call stacks between suspicious methods and raw-memory accesses is 9.

We make several observations from the study. First, for our subjects, the technique successfully clusters patterns. The clusters included *all* 24 of the true bugs across all subjects, with just 1 false positive cluster. Thus, for each bug, a developer would typically need only to investigate one bug graph to understand the bug in a high level, and investigate only up to 20 patterns to understand the bug in a low level. However, without such clustering, a developer might need to investigate an indefinite number of patterns—up to 71k patterns as shown for `Mysql-791` in Table 11.

For the subjects we investigated, we found only one false positive in `Mysql-169`. The bug is a multi-variable atomicity violation.[5] Our manual investigation found that the bug graph for the true-positive cluster represents the real bug as given in the bug description. However, the bug graph for the false-positive cluster shows two thread-context changes, where the context changes resemble those of the real bug but differ slightly in that the accesses in the false-positive cluster were observed closely in time. (The details of this bug are given at the link in Footnote 3.)

Second, for our subjects, we found that all suspicious methods locate the method that is the cause of bug. For `Vector-1` and `Vector-2`, suspicious methods include the `Vector` constructor. For another example, for `Transmission`, GRIFFIN reports `tr_sessionInitFull` as the suspicious method, where the incorrect ordering actually occurs.[6]

[5] `http://web.eecs.umich.edu/~jieyu/bugs/mysql-169.html`
[6] `http://web.eecs.umich.edu/~jieyu/bugs/transmission-142.html`

Third, for our subjects, call-stack sizes between suspicious methods and raw-memory accesses are greater than zero except for `PBZip2`. Considering the large call stack sizes in Table 12 and the large number of patterns in Table 11, we can confirm that Challenges 2 and 3 in Section 5.2.3 are not trivial.

5.4.5 Threats to Validity

There are several threats to validity of our studies. Threats to internal validity include the empirical setup, especially the test suite that we used for our studies. Because there is no standard benchmark suite for use in fault-localization of concurrency bugs, we used programs with known bugs. However, to mitigate this threat, we created and used the test suites as described in the bug description of the repositories to apply our technique to test suites of open-source programs.

Threats to external validity limit the generalization of the technique in real debugging situations. Because of the lack of a standard benchmark suite, this problem is common to all existing work in the area. However, to reduce this threat, we chose programs with real bugs, and we used programs for both Java and C++.

Threats to construct validity include the way developers utilize the outputs of the fault-localization techniques. Our technique provides three different forms of output, so developers may understand bugs in different views with our results. In addition, we evaluated our technique with a set of experiments, showing that our technique pinpoints bugs in both method and memory-access levels. However, we acknowledge that only user studies will inform us about the usefulness and effectiveness of our techniques.

5.5 Related Work

This section discusses existing fault-localization and fault-fixing techniques, along with fault-clustering and fault-comprehension techniques, and compares them with Griffin.

Fault localization. Early work on fault-localization techniques for concurrent programs located one type of concurrency faults. These techniques statically inspect program code [22, 60] or dynamically monitor shared-memory accesses [25, 90], and report a list of problematic memory-accesses patterns representing data races [60] or atomicity violations [22, 25, 90]. More recent work investigates memory-access patterns for several types of concurrency bugs and outputs these patterns as bug reports [34, 54, 68, 83]. All these fault-localization techniques report a list of memory-access patterns as bugs. However, Griffin clusters these patterns for each bug and provides further contextual information to improve comprehension of concurrency bugs.

Automatic fix. There are a few recent techniques for automatic fixing of concurrent programs [33,35,44]. AFix [33] and AXis [44] input atomicity violations from the output reports of existing techniques [29], determine the atomic regions from the reports, and insert locks to protect atomic regions. CFix [35] extends AFix to handle both atomicity and order violations. Griffin differs from these techniques in two ways. First, the goal of Griffin is to enhance understanding of bugs by providing clusters of patterns and suspicious method calls, and to let developers fix the bug with various strategies. In contrast, these fix techniques assume complete understanding of bugs, and use simple strategies such as lock-insertion, that developers rarely use [50]. Second, Griffin can help automatic fix techniques because it provides a more complete explanation of bugs, which can be input to these techniques.

Fault clustering. One of the main parts of Griffin is clustering failing executions, and there are largely two types of techniques for the clustering as discussed in Section 5.3.2: profile-based clustering and fault-localization-based clustering. Podgurski and colleagues [72] showed that program profiles can be used for clustering failing executions according to the causes of the faults. Zheng and colleagues [98] presented a profile-based clustering algorithm that finds bug predicates and that handles multiple

bugs. Jones and colleagues [37] utilized both profile-based and fault-localization-based clustering to improve both sequential and parallel debugging, and showed that both algorithms save debugging cost over non-clustered techniques. Our technique uses a fault-localization-based clustering technique because fault-localization data is available from an existing technique. However, unlike these techniques, our technique also uses a pattern clustering algorithm to further refine the clustering results.

Fault comprehension. There are several approaches that improve understanding of bugs by providing additional information over fault-localization results for sequential programs. The underlying idea of the approaches is that providing suspicious statements is not enough to identify and understand the bug, and that providing additional information (e.g., clustering results and explanations) can improve developers' understanding [71]. Rapid [28] provides suspicious statements with the method context leading to the statement so that developers can infer the execution path to the bug. LEAP [12] provides suspicious graphs at the method and the basic-block levels using graph-mining algorithms so that developers can understand the bug at a level higher than statements. Like these approaches, GRIFFIN provides additional information over fault-localization techniques. However, unlike these techniques, GRIFFIN works for concurrent programs.

5.6 Summary

In our view, GRIFFIN makes significant progress toward filling an important research gap in *fault-comprehension*, which lies between fault-localization and fault-fixing techniques for concurrent programs. Recall that, to date, fault-localization has focused on pinpointing basic program fragments (memory references) that lead to bugs, and fault-fixing techniques attempt (semi-)automatic repair through the use of simple patches. However, experience suggests that true fixes require deeper program understanding, which GRIFFIN attempts to provide through explicit identification of

suspicious *methods* rather than just memory references; reconstruction of *bug context* through analysis of call stacks, rather than just reporting call stacks; and the use of bug graphs to aid *developer* understanding of this information.

CHAPTER VI

EMPIRICAL USER STUDY OF DEBUGGING TECHNIQUES

6.1 Introduction

Although Falcon, Unicorn, and Griffin aim to improve the understanding of concurrency bugs, this aim needs empirical testing. This chapter summarizes an empirical user study whose goal is to observe how programmers use the information provided by these and a related tool to understand and fix concurrency bugs. In particular, we implemented a baseline technique with two debugging techniques, Unicorn and Griffin, in Eclipse plugins and designed an empirical user study in which we assigned different tools for developers for different levels of debugging tasks. Then, we observed the debugging behavior of the participants, surveyed their experience, and analyzed the responses quantitatively and qualitatively.

The results are interesting in several aspects. For example, Griffin is more useful for understanding concurrency bugs especially for harder tasks. We also found that developers use the tools in different ways, e.g., to track the root cause of the bug or to confirm the root cause after finding it with manual source inspection. Moreover, users suggested other improvements, such as suggestions for enhanced visual representation of the tools.

The main contributions of this empirical user study are summarized as follows:

- An empirical user study that determines whether the debugging tools for concurrency bugs help developers.
- Analyses of the study results, which partially validates the effectiveness of Unicorn and Griffin.

- Discussions of our experience from the empirical user study.

6.2 Related Work

This section discusses empirical user studies for evaluating debugging techniques for sequential and concurrent software.

6.2.1 Empirical User Studies for Sequential Bugs

Several types of debugging techniques for sequential bugs have been presented over the past three decades. Weiser proposed one of the first debugging techniques, called program slicing [91]. Their main idea is that developers can investigate only "slices" of program behavior. Another debugging approach is statistical fault localization [39, 42], where these techniques identify potentially suspicious statements from multiple failing executions and often compare the failing executions with passing executions. Other approaches help developers in debugging software by allowing them to ask "why" questions on program behavior [40].

These techniques have been evaluated with real programmers and students. Weiser has evaluated the effectiveness of program slices in debugging 100 lines of programs with 21 programmers, and found that programmers tend to follow the program execution flow [91]. Parnin and Orso evaluated the effectiveness of statistical fault localization techniques in debugging programs of 2KLOC and 4KLOC with 34 graduate students. The results show that the techniques help expert programmers debugging programs, but the techniques need to provide some more information, e.g., context, to be more effective [71]. Finally, Ko and Meyers evaluated the WhyLine tool with 10 graduate students [41], and found that WhyLine is effective in debugging sequential programs both in time and completion of the debugging.

6.2.2 Visualization Tools for Concurrent Software

As an aid for debugging, visualization tools display traces with focus on the interactions among threads [1,7,55,74,75]. For example, model checkers, such as JPF [26] or CHESS [1], produce a serialized execute trace when a concurrent program fails. Concurrency Explorer [1] takes a trace produced by CHESS and displays the sequence of thread interactions with a list. TIE [55] extends Concurrency Explorer and displays traces from multiple executions. Artho et al [7] presents the interactions of threads as a UML sequence diagram to express the bug with the clear interactions. JIVE and JOVE [74, 75] focus on incurring minimal instrumentation overhead for visualizing threads for Java programs.

We believe an empirical user study to evaluate these techniques would be useful for the following reasons. First, most techniques were developed only for research, and they were not formally evaluated on human subjects [7, 55, 74, 75]. Second, except Concurrency Explorer, which is used internally in Microsoft, the other tools were evaluated with toy programs, not with any real programs. Thus, it is not clear whether the visualization tools are scalable for real concurrency bugs as the trace size increases.

6.2.3 Empirical User Studies for Concurrency Bugs

There are several empirical user studies on how programmers develop concurrent programs [15, 16, 27, 45–47, 62, 87]. These studies evaluated several aspects of novice and experienced programmers, such as how much speedup they achieve, how concisely they write programs, how they design programs, and so on. However, the studies do not evaluate on how developers debug concurrency bugs or how developers use debugging tools for concurrency bugs.

To our knowledge, there are only two related works for debugging concurrent programs. The first work is Lönnberg et al's studies on how students understand

concurrency bugs [48]. They performed several empirical user studies on students, by asking the students to write concurrent programs and interviewed them several questions [45, 46]. They then suggest several ways to help students in debugging concurrent programs. For example, they claim that students usually have different understanding of concurrency from teachers, and thus software visualization tools will help both teachers and students get the same view of the programs and bugs. However, the authors mainly suggest how one might design such visualizations, without providing user evaluation.

The second work is Sadowski and Yi's user evaluations on how developers use the new concurrency notation, called *cooperability* [79]. They posted three concurrency bugs on an internet-based survey form, divided participants into two groups, where one group of people have an aid of cooperability and the others do not, and evaluated the responses. They scored the correctness of the responses with the ranking scheme and statistically showed that developers can understand concurrency bugs better with the aid of cooperability. However, cooperative programming is a concurrent programming scheme, but not a debugging technique, so this study is not an evaluation on debugging tools for concurrent software.

To our knowledge, there are no user evaluations on automated concurrency bug debugging tools. Because of this lack of information, existing tools were based on bug characteristics, not based on developer's need. In our study, we evaluate whether existing tools can assist developers in understanding concurrency bugs.

6.3 Goal and Hypotheses

The main goal of our empirical user study is to evaluate whether fault-localization techniques for concurrent programs can help in understanding concurrency bugs. Thus, we designed a study to compare fault-localization techniques (UNICORN and GRIFFIN) with a baseline tool, called TRACER (See Section 6.4.2 for TRACER).

The central assumption of UNICORN is that providing likely buggy memory-access patterns will help locate the concurrency bugs, and so will increase the understanding of the bug. Based on the assumption, we can pose the first hypothesis.

Hypothesis 1: Participants who use UNICORN will understand concurrency bugs better than participants who use TRACER.

GRIFFIN was developed to address problems in UNICORN, such as memory-accesses are not enough to explain concurrency bugs and more contextual information is needed (Section 5.2). Thus, in our study, we will investigate whether participants perform better with GRIFFIN, and we pose the second hypothesis as follows.

Hypothesis 2: Participants who use GRIFFIN will understand concurrency bugs better than participants who use UNICORN or TRACER.

Finally, we assumed that if UNICORN and GRIFFIN assist developers in understanding concurrency bugs, the developers will continue to perform well in fixing concurrency bugs as well. Thus, our last hypothesis is as follows.

Hypothesis 3: Participants who use either UNICORN or GRIFFIN will fix concurrency bugs better than participants who use TRACER.

A key question is how one measures "understanding". Essentially, our approach is to ask the user a series of questions about the nature of the fault, and to grade these responses. For details, see Sections 6.4.6 and 6.4.7.

6.4 Experimental Protocol

6.4.1 Program Subjects

We selected program subjects that were used in our previous studies with the following policies. First, we selected only Java subjects and excluded C/C++ subjects, so that participants can focus on programs written in one language. Second, we selected

non-trivial programs with different sizes. Thus, we selected one small-sized program (Bank Account), one medium-sized program (Shop), and one large-sized program (List). Third, we did not consider programs with "huge" traces. In particular, we did not select several subject programs because they generate traces that the baseline tool (TRACER) cannot load.

6.4.1.1 Bank Account

`Bank Account` is a program included in the Contest Benchmark Suite, which consists of Java programs with seeded concurrency bugs [18]. The program consists of 116 LOC including comments and blanks. On startup, the program creates multiple account owners as individual threads and executes them concurrently. Each account owner initially has $300, deposits $100, withdraws $100 from her own account, and transfers $100 to the next owner's account. When the program executes successfully, each account owner should have $300 in her account in the end. However, when one owner's transfer is executed concurrently with another owner's transfer, the final amount may be some different values, such as $200 or $400, not $300.

6.4.1.2 Shop

`Shop` is another program included in the Contest Benchmark Suite [18]. The program consists of 296 LOC including comments and blanks. This program is a classic example of the producer-consumer problem. There are three major objects, Shop, Supplier, and Consumer, in the program. In the beginning, a Shop object is initialized, and it is passed to a Supplier thread and multiple Consumer threads. The Supplier thread generates items using the Shop object, and multiple Consumer threads take the items from the Shop objects. If two Customer threads take items concurrently when there's only one available item, the program can fail with an `ArrayIndexOutOfBoundsException`.

6.4.1.3 List

`List` is a data structure included in the Java Collection Library.[1] We extracted the library from the JDK 1.4 source, and the total size is 25,814 LOC. Because it is an open program, we created a test harness, which can fail with a concurrency bug. The program initially creates three SynchronizedList, initiates four threads that share these three `SynchronizedLists`, and executes the threads concurrently. Each thread executes a method from the List class (e.g., `add`, `addAll`, `remove`, and `copy constructor`) for the three shared objects. When the copy constructor is executed, the passed object is changed concurrently in the other thread, the copy constructor can take a null item in the list.

6.4.2 Debugging Tools

As our fault-localization techniques, we chose UNICORN and GRIFFIN for the following reasons. First, UNICORN and GRIFFIN have distinct features—UNICORN reports likely buggy memory-access patterns, whereas GRIFFIN reports grouped memory accesses with calling contexts, so their report type is quite different. Second, we did not choose FALCON because both FALCON and UNICORN report memory-access patterns and the coverage of UNICORN subsumes that of FALCON.

To compare UNICORN and GRIFFIN, we selected a baseline technique. Because there are no industrial automated debugging tools for concurrency bugs, we selected a visualization tool, ConcurrencyExplorer [1], among many visualization tools [1, 7, 55, 74, 75]. We chose ConcurrencyExplorer over other tools because it is an industry tool used in Microsoft, and it presents the execution information in an intuitive manner.

We implemented the tools in Eclipse.[2] The following subsections discuss the implementations in detail.

[1] `http://www.agent31.eu/2009/06/hidden-java-concurrency-bugs.html`
[2] `https://www.eclipse.org/`

6.4.2.1 Tracer

TRACER is our Eclipse-plugin implementation of ConcurrencyExplorer [1]. Since ConcurrencyExplorer is implemented for Visual Studio, so we implemented TRACER in Eclipse for Java programs. Like the original tool, our tool shows a memory dump of failing program execution. For our study, we instrumented the programs, ran them, got memory dumps from a failing execution, and showed the dumps using our plugin.

Figure 13 presents a snapshot of TRACER. The list view shows the sequence of executed memory accesses. Each row represents a memory access, which consists of the global access index, thread ID, memory access location in the source code, source statement, access type, and its parent method. For example, the first row in Figure 13 shows the first shared memory access in a failing execution, where it is a read access on variable, `c`, in the method, `increment()`, in line 5 of Counter.java. To help developers identify accesses from different threads, the accesses from the same thread are showed with the same color. The top right shows a thread-selector view, which contains all execution threads with combo boxes. It allows the participants to selectively see accesses from specific threads. When the participant double clicks each row in the list view, the editor opens the related source code file and line. The plugin records all participant's activities, including clicks and key strokes in the tool and editors for evaluation purpose.

Note that, because the tool shows a complete dump of an execution, as the program execution becomes larger, the tool may have to show a very huge information. Thus, we excluded some programs for our subjects because their execution dump was too huge to show for our empirical user study (more than 10K memory accesses).

6.4.2.2 Unicorn

Our UNICORN technique, presented in Chapter 4, is implemented as an Eclipse plugin. We instrumented each subject program, ran it with the tool, got suspicious memory

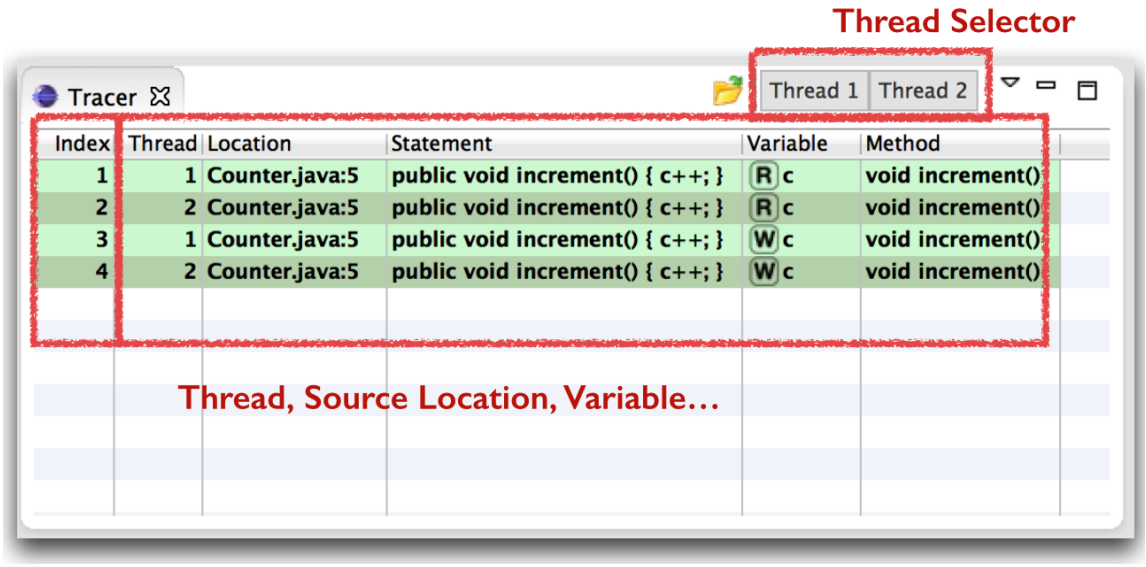


Figure 13: Eclipse plugin for TRACER.

accesses, and showed the results with our plugin.

Figure 14 shows a snapshot of UNICORN. To minimize the confusion of using different tools, we maintain a similar look-and-feel with TRACER, and so we implemented the tool with a tree view. Each element in a tree view represents a memory-access pattern, and the elements are ordered by suspiciousness scores. The memory-access pattern consists of its pattern type, the summary of the patterns in the method level, and the actual memory accesses in two threads. For example, the 1st ranked item in Figure 14 is a RWW pattern, where the three accesses appear in `increment` method in two threads, and they are accessed in line 5 of Counter.java. Unlike TRACER which shows accesses from all threads, this tool shows the interaction of any two threads and accesses from the same thread are showed with the same color. Like TRACER, when the participant double clicks each row in the list of accesses, the editor opens the related source code file and line. The plugin records all participant's activities, including clicks and key strokes in the tool and editors for evaluation purpose.

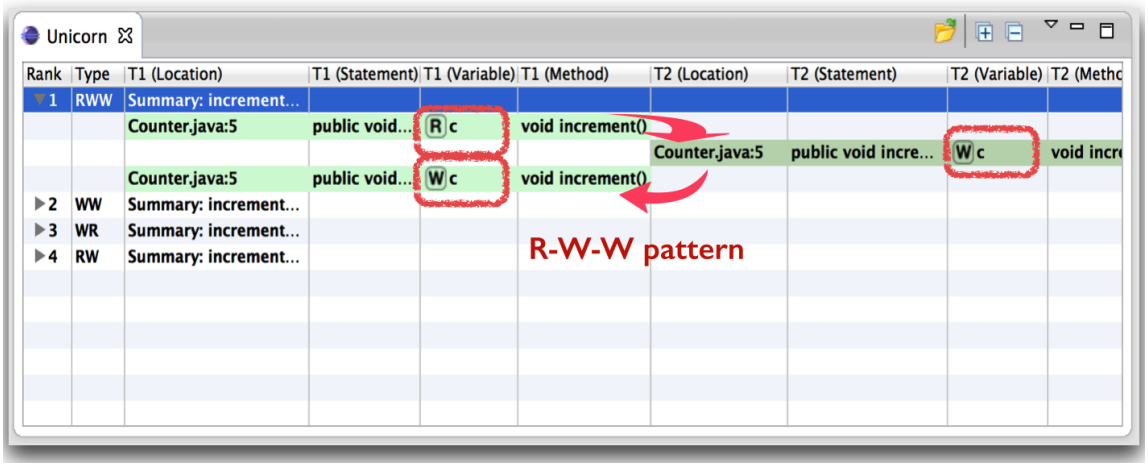


Figure 14: Eclipse plugin for UNICORN.

6.4.2.3 Griffin

Our GRIFFIN technique, presented in Chapter 5, is also implemented as an Eclipse plugin. We instrumented each subject program, ran it with GRIFFIN technique, got groups of memory accesses with their calling context, and showed the results with our plugin.

Figure 15 shows a snapshot of GRIFFIN. To maintain a similar look-and-feel with TRACER and Unicorn, we implemented the tool with a tree view. Each element in a tree view represents a group of memory accesses with calling context. Like Unicorn, it shows interleaving of any two threads. For example, the 1st item in Figure 14 is a set of memory interactions in T1, followed by T2, again by T1. Unlike Unicorn, it provides more contextual information for concurrency bugs, such as clustered memory accesses (instead of memory-access patterns up to four accesses), suspicious methods, and calling contexts. Like TRACER and Unicorn, when the participant double clicks each row in the accesses and calling contexts, the editor opens the related source code file and line. The plugin records all participant's activities, including clicks and key strokes in the tool and editors for evaluation purpose.

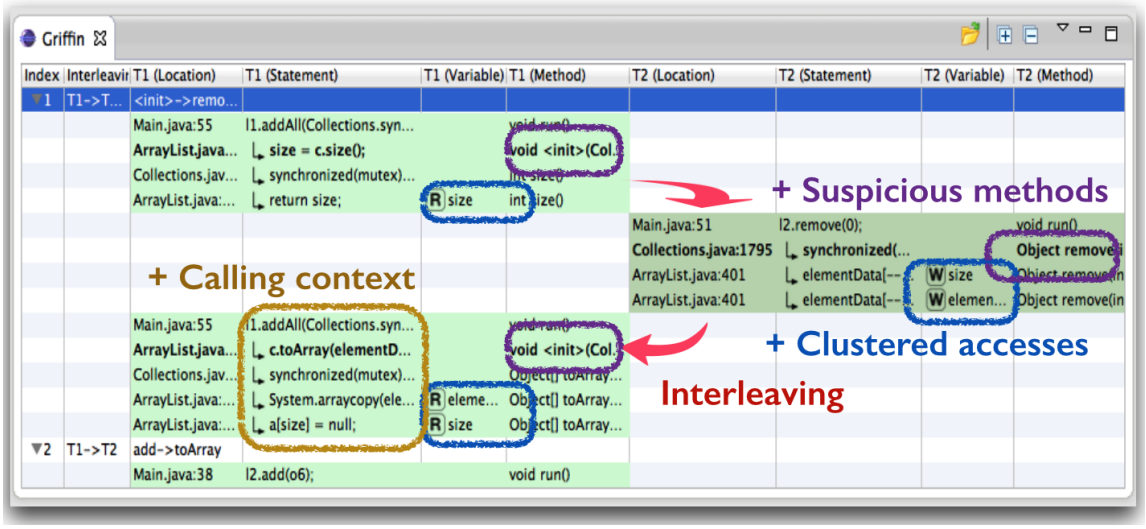


Figure 15: Eclipse plugin for GRIFFIN.

6.4.3 Tasks

We gave each participant three main tasks, where the goal of each task is debugging a concurrency failure in a program subject. For each task, the participant was provided with a description of the subject program, the result of a passing execution, the result of a failing execution, and information on how to run the subject program. After reading the description, the participant was instructed to navigate the source code and identify the root cause using any functionality of the Eclipse IDE (e.g., search with keyword or see class hierarchy) or by using the tool. In addition, she could run the program to see the results. However, due to the non-deterministic nature of concurrency bugs, we made her aware of the difficulty in reproducing them. We gave the time limit to participants to identify the fault, and then they completed a questionnaire, which asked them to indicate the root cause of the fault and a possible fix. We instructed them not to fix the fault in Eclipse, but to write about the fix in the survey questionnaire.

The complete descriptions of the subject programs, successful execution logs, and failure execution logs are presented in Appendix A.

6.4.4 Participants

We recruited graduate students in Computer Science at Georgia Tech as participants for our study. The participants had at least some multi-threaded programming experiences. They had 2 to 30 years of programming experience with a median of 11; They had 1 to 20 years of Java experience with a median of four; and, *all participants* were familiar with concurrent programming from a class project, an industry/research project, or multiple projects. As the participants ranged from beginners to experts in concurrency, we randomly assigned them to assess how the debugging tools could help different classes of programmers.

6.4.5 Study Design

We used factorial design for our study [2]. The main reason for choosing this study design was to collect the developer's impression on using different tools. Specifically, for three tools and three subjects, we made six different combinations ($3\times2\times1$), and assigned the participants in one of the six groups. Each participant used different tools for conducting different tasks, and at the end of completing all tasks, the participant were asked to rank the effectiveness of the tools.

6.4.6 Procedure

Participants performed the study in an office at Georgia Tech. We setup three computers with a Linux virtual machine, on which subject programs and Eclipse tools are installed. When a participant entered the room, the experimenter gave her the consent document and discussed the instructions for the study. Our instructions included Java concurrency tutorial, examples of types of concurrency bugs and related fixes, and demo of all three Eclipse tools. Then, the experimenter randomly assigned her a group from the factorial design and issued a random four-digit number as unique user identifier (UUID). The participant used the UUID throughout the survey questionnaires so that their identities were anonymized. Then, the participant completed a

background survey, which asked their programming experience, and completed each task for 20 minutes. The survey questionnaire for each task included the usefulness of the tool, the root cause of the fault in four questions (i.e., type of the bug, problematic variable, problematic method, and the description of problematic interleaving), and the fix suggestion in two questions (i.e., type of fix and the detailed description of it). Finally, the participant completed a final survey questionnaire, which askd the comparison of the tools and their overall experience. The complete list of the survey questionnaire is in Appendix B. We allocated 1 hour and 30 minutes in total, which consists of 20 minutes for instruction, 1 hour for completing three tasks, and 10 minutes for the final survey.

6.4.7 Evaluation Method

We collected several data for each task—completion time, Eclipse navigation data, usefulness score, understanding score, and fix score. To test hypotheses, we compared the above data for each task and tool pair using statistical hypothesis testing method.

We manually graded the understanding score with the following scheme. If the participant wrote "I don't know" or a completely wrong description, we gave the score of 1. Otherwise, we set the base score to 5 and deducted -1 for each of the followings: wrong type of bug, wrong variable name, wrong method name, and wrong description of interleaving. Thus, the score becomes 1 (low) to 5 (high).

We manually graded the fix score with the ranking scheme [4]. We could not apply the same grading scheme used for computing understanding score because the participants gave written descriptions of the fix, which usually did not indicate the specific locations to be modified. Thus, it is difficult to add or deduct points from the written description. In summary, we reviewed all answers and ordered them from bad descriptions to good descriptions, and gave them scores from 1 (low) to 5 (high) scale.

We calculated the numbers of activities from Eclipse logs. In addition, we manually inspected the navigation data to investigate participant's activities for special instances.

6.4.8 Data Availability

The virtual machine, Eclipse plugins, subjects, instructions, and survey forms are publicly available at `http://www.cc.gatech.edu/~sangminp/concurrency-study/`.

6.5 Results

We conducted the study for 32 students. Before analyzing the results, we removed two outliers, who scored 1 point in at least two tasks, because they lacked enough expertise to do concurrent programming. Then, we computed understanding and fix scores and performed hypothesis testing.

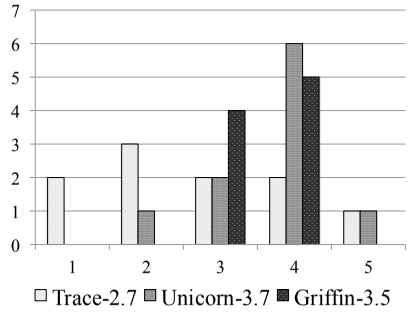
The overall results show that Hypothesis 1 does not hold, but Hypothesis 2 holds for difficult tasks. More specifically, participants understood the bugs significantly better when using Griffin for hard tasks. Another result shows that Hypothesis 3 does not hold for all tasks.

We show the overall score distribution in Figure 16 and hypothesis testing results (unpaired t-test) in Table 13, and discuss the significance of the results in the following section.

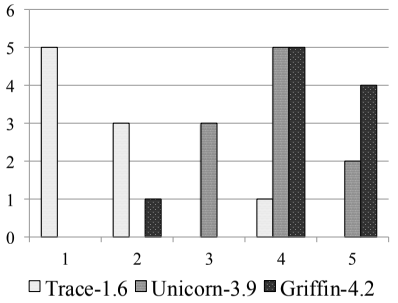
6.5.1 Overall Results

In this section, we investigate the differences of usefulness, understanding, and fix scores. Recall that participants gave a usefulness score in 1 to 5 scale, and we graded the understanding and fix scores with our grading scheme (Section 6.4.7). We did not analyze navigation data and completion time, but discuss them later in this section.

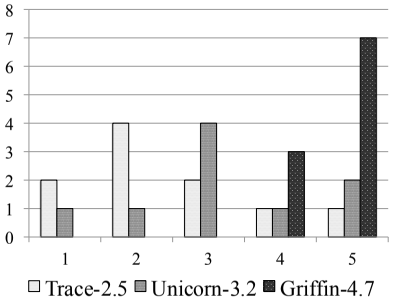
The plots (a) to (i) in Figure 16 show the distributions of scores. The horizontal axis presents the scores in 1 (low) to 5 (high) scale. The vertical axis presents the



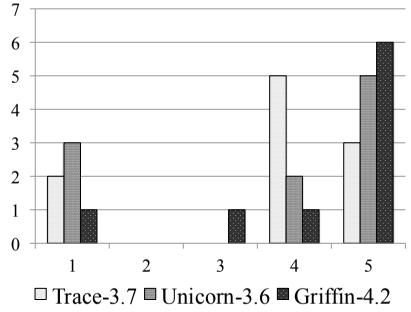
(a) Usefulness (Task 1: Bank Account)



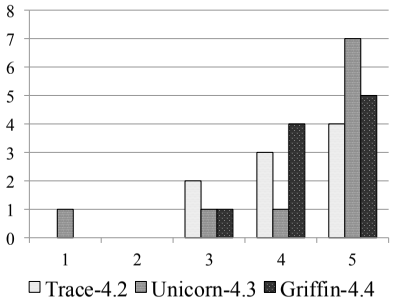
(b) Usefulness (Task 2: Shop)



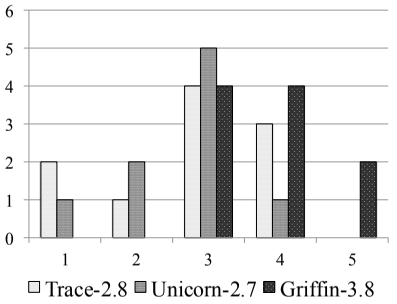
(c) Usefulness (Task 3: List)



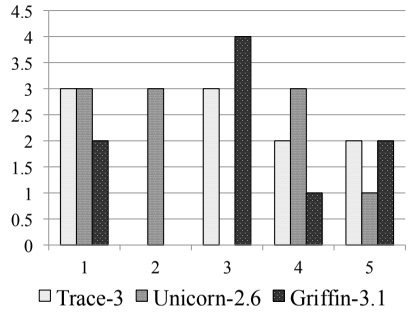
(d) Understanding (Task 1: Bank Account)



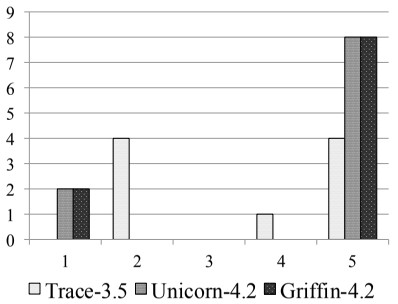
(e) Understanding (Task 2: Shop)



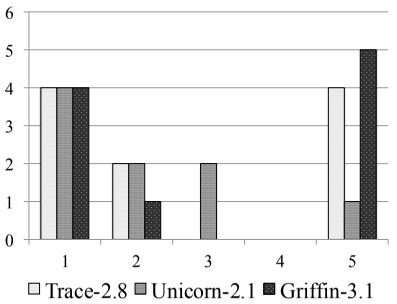
(f) Understanding (Task 3: List)



(g) Fix (Task 1: Bank Account)



(h) Fix (Task 2: Shop)



(i) Fix (Task 3: List)

Figure 16: Overall score distribution.

number of participants. There are three legend entries for each plot. Each entry is composed of the tool name and the average. For example, plot (a) presents the distribution of usefulness scores for Bank Account task. The participants using Tracer, Unicorn, and Griffin got average scores, 2.7, 3.7, and 3.5, respectively.

Table 13 shows hypothesis testing results. The numbers in the cells represent the mean differences. The colored cells represent that the hypothesis testing result is statistically significant (i.e., $p < 0.05$). For example, the second row shows the test results for comparing usefulness of Griffin and Tracer. For Task 1, there's no difference in the usefulness because the result is not statistically significant, but for Task 3, Griffin is more useful than Tracer because the result is statistically significant with mean difference 2.17.

For Task 1, the distributions of the scores are in plots (a), (d), and (g) in Figure 16. We found that the differences are not statistically significant as in the third column of Table 13 although Griffin and Unicorn has higher mean values in most cases. Plot (d) explains this result. Because Task 1 was the easiest task, most participants understood the root cause well regardless of the tool and did not differentiate the usefulness of the tools. Plot (g) explains that there is any correlation among different tool users. Thus, Hypotheses 1, 2, and 3 do not hold for Task 1.

For Task 2, the distributions of the scores are in plots (b), (e), and (h) in Figure 16. We found that the usefulness scores are statistically significantly different for comparing Griffin and Unicorn with Tracer as in the fourth column of Table 13. The results indicate that, for the medium sized subject in our study, participants experienced that Griffin and Unicorn are more useful than Tracer. However, understanding and fix scores are not statistically significantly greater. We interpret this result as follows: Since the subject is of medium difficulty, with the help of the tool, the participants could easily find the bug; however, they apparently found the bug with manual inspection. Thus, Hypotheses 1, 2, and 3 do not hold for Task 2.

Table 13: Hypothesis testing results.

Score Type	Hypothesis Testing	Task 1: Bank Account	Task 2: Shop	Task 3: List
Usefulness	Griffin > Tracer	0.67	2.53	2.17
	Griffin > Unicorn	-0.14	0.31	1.44
	Unicorn > Tracer	0.81	2.22	0.72
Understanding	Griffin > Tracer	0.96	0.18	0.98
	Griffin > Unicorn	0.62	0.07	1.11
	Unicorn > Tracer	-0.07	0.11	-0.13
Fix	Griffin > Tracer	0.24	0.64	0.42
	Griffin > Unicorn	0.51	0.09	1.11
	Unicorn > Tracer	-0.29	0.56	-0.69

For Task 3, the distributions of the scores are in plots (c), (f), and (i) in Figure 16. We found that the usefulness and understanding scores are statistically significantly different for comparing Griffin with Unicorn and Tracer as in the fifth column of Table 13. The results indicate that, for the most difficult task, participants experienced that Griffin is a more useful tool than others, and they also understood the bug better. However, we also found no significant difference of Tracer and Unicorn. Regarding the fix score, we did not find any correlation among different tool users. Thus, Hypothesis 1 does not hold, but Hypothesis 2 holds for Task 3.

Overall, here are the hypothesis testing results. First, we **did not find strong support for Hypotheses 1**, i.e., either Unicorn does not help developers in understanding concurrency bugs, or the experiment was insufficient to confirm the claim. It is interesting that participants found that Unicorn is useful only for medium task. We believe that this happens because memory accesses may help developers in confirming concurrency bugs for medium task, but memory accesses without contextual information do not help developers for difficult tasks.

Second, we found **support for Hypothesis 2 for difficult tasks**. For medium and large tasks, participants found that Griffin is useful, and for the most difficult task, they actually understood better with Griffin. We found several supporting

comments and discuss them in Section 6.7.

Third, we **did not find strong support for Hypothesis 3**. That is, we did not find strong support that the tools help developers for fixing concurrency bugs. We interpret this result as follows: the tools are not designed for providing information for fix, but providing information for fault localization. Thus, there are no correlation between high scores of fix and high scores for understanding. We believe that participants wrote their potential fixes based on their prior concurrency experience without any guidance from the tools. We discuss this observation with supporting comments in Section 6.7.

We investigated navigation data and completion time. Regarding the navigation data, we found that, for Task 3, participants had much fewer average Eclipse activities when using Griffin (39.6) than when using Unicorn (55.3) or Tracer (58.7). We interpret this result as follows: participants did not have to navigate much using Griffin because they found the bug early. We did not find any interesting navigation data for Tasks 1 and 2. It is because the tool was designed to improve the understanding of the bug by presenting information about the bug, but not designed for reducing the number of activities, e.g., by providing step-by-step clicks to track the bug.

We did not find any interesting observation in completion time because most participants used the time limit to complete the tasks.

6.5.2 Results by Tool Preference

In this section, we find the most preferred tool and investigate the reason. To to the analysis, we collected the results of the tool preference questionnaire in the final survey and associated the results with the understanding and fix scores. Specifically, we assigned the users to Group-T, Group-U, and Group-G, for participants who rated Tracer, Unicorn, and Griffin as their best tools, respectively, and we computed

Table 14: Average score by tool preference.

Task	Score Type	Group-T (2)	Group-U (7)	Group-G (21)
Task 1: Bank Account	Understanding	3.0	3.75	3.78
	Fix	2.0	2.37	3.05
Task 2: Shop	Understanding	3.33	4.12	4.26
	Fix	2.33	3.75	4.0
Task 3: List	Understanding	2.66	2.75	3.05
	Fix	1.33	2.87	2.68

the average understanding and fix scores for each group.

Table 14 shows the results. The first column shows the task type; the second column shows the score type; and, the final three columns show the average understanding and fix scores. The numbers in the header of the final three columns show the number of the participants. Thus, the number of participants of Group-T, Group-U, and Group-G are 2, 7, and 21, respectively. For example, for Task 1, Group-T, Group-U, and Group-G got 3.0, 3.75, and 3.78 average understanding scores, respectively. Note that, we do not perform hypothesis testing because the sample size of Group-T is only two.

One interesting observation is that Group-G consists of 21 participants, which is 70% of all participants. This observation indicates that Griffin is the most preferred tool for debugging concurrency bugs. In addition, we observed that participants in Group-U and Group-G understood and fixed concurrency bugs better than participants in Group-T with higher average scores.

Another observation is that the size of Group-T is only two, which is 7%, and the overall performance of Group-T is worse than that of the other groups. We investigated the comments of the two participants in Group-T to understand why they prefer Tracer to other tools. They answered that Tracer is better because it completely shows the whole execution context. However, because the tool does not provide any summary of concurrency bugs, the participants did not understand and

fix concurrency bugs well.

6.6 Limitations

Our study has multiple limitations, which future work can address. The first limitation involves the size of the participants. Although we recruited 32 students, we believe that more participants will make the results more conclusive.

The second limitation involves the technical expertise of the participants. We mainly recruited graduate students with the following reasons. Most graduate students are skilled programmers with many years of programming experience (10.8 years on average). On the other hand, professional developers have different ranges of experience and their concurrency experience might vary a lot. Since the tools are to help professional programmers, the ultimate goal of such a study is to recruit such expert developers. Thus, a future user study can focus on recruiting professional developers.

The next few limitations are related to the selection of our subject programs. The third limitation involves the complexity of the subject programs. `Bank Account` has 100 lines of code, and the concurrency bug may seem to be too easy for developers to debug. Thus, one may argue that it is not clear whether we might learn anything from this subject program. However, an easy subject is good to start with because it helps calibrate the session for participants. In addition, we analyzed the subject separately and found that the three tools are not distinguishable for the easy subject case, which confirmed our expectations.

The fourth limitation involves the fact that participants of the study were asked to debug code written by other developers. One may claim that this situation is rare. However, if true, we would have found a positive correlation between "understanding" and "fixing". This reinforces the need to spend more time studying the difference between those two aspects of debugging in the future.

The fifth limitation involves the scalability. One may argue that the tools should

be tested against large programs with millions of lines of code and hundreds of threads. Since our study investigates the effectiveness of the tool for small numbers of threads (i.e., up to five threads in `List`), the scalability limitation can be investigated in a future study.

The sixth limitation involves the baseline tool selection. One possible baseline might be simply to ask participants what they would normally do. However, since we have more narrow experimental setup, which includes an uniform UI, we can measure the specific effects of certain kinds of information.

The seventh limitation involves the test input we provided. We wrote our test inputs mainly to trigger concurrency bugs so that developers can focus on tracking concurrency bugs using the input. However, in the real world situation, a developer may spend most of her time in minimizing the test input to narrow down a specific package in the source code and spend less time on tracking concurrency bugs in the package. Future work may investigate the effect on test minimization.

The eighth limitation involves the factorial study design. We used this design in order for participants to rank the three tools. However, the participants may not rank the tools objectively because they used different tools for different tasks. If we had used a different study design, such as within-subject method [5], a participant would have used all tools for all tasks and might objectively judge the tools. We did not use this method because participants are biased once they find a bug.

The final limitation is the stability of the results and the reproducibility of the study. For instance, the sample size is small, and the grading involves manual inspection of the answers. To mitigate this limitation, we released the results in public.

6.7 Discussion

This section discusses our findings from the observations on developer's behaviors and their comments from the study.

6.7.1 Experience on The Tools

6.7.1.1 Griffin and Unicorn are more useful for hard tasks

Our results showed that GRIFFIN is useful for medium and hard tasks. We assumed that GRIFFIN is useful because it provides more contextual information of the root cause. Many participants confirmed this assumption with supporting comments. One interesting comment is as follows:

"*There are three dimensions to think about: Time vs. Threads vs. code's stack. Griffin showed these three dimensions quite effectively. However, the other two tools provide only some of them.*"

In addition, we found that the ranking system did not have a significant effect to identify the bug, and the ranking scheme of UNICORN and GRIFFIN are still useful. This observation corresponds to the observation from the study of the fault localization tools for sequential programs [71]. When using UNICORN and GRIFFIN, participants investigated several entries at the top in the list and found the bug without any complaints:

"*Though the tool provided false positive as a #1 candidate, it was still useful to diagnose the error without going deep into the code.*"

6.7.1.2 Tracer needs improvements

We assumed that TRACER is not so useful because it does not provide the summary of concurrency bugs, so the participants might be overwhelmed by too much information. Indeed, most participants performed worse with TRACER and confirmed our assumption with supporting comments. Two interesting comments are as follows:

"*Tracer might be used for simple code. However, overall I don't think it can be useful in a real life scenario because most codes are likely complicated.*"

"*The tool wasn't very useful on this task because there were too many threads and instructions to keep track of.*"

It is noticeable that the participant of the first comment expected that TRACER might not be scalable. The problem happened in our program subject selection, and we also expect that TRACER is only useful for small subject programs.

6.7.1.3 The tools are utilized in multiple ways

We found that developers use the tools in multiple ways. First, some developers used the tools to *track* using the information given by the tool. When a participant starts a task, she clicks the entries in the list of the tool to inspect the code around the bug. If they want some more contextual information, they would inspect some more program code on demand.

On the other hand, other developers used the tools to *confirm* the bugs using the information by the tool. On startup of a task, these developers ran the program several times to understand how the program works and inspected the code from the main entry. Then, they manually find the bug from code inspection, and they investigated the result of the tool to confirm whether their finding was right. Here is one comment:

"*The tool was very useful in confirming the bug; I used code inspection to determine the bug.*"

This comment was for GRIFFIN and thus we found that GRIFFIN was useful in both cases.

6.7.2 Improvements

6.7.2.1 The tools need better visualization support

Many participants complained that the current implementations of UNICORN and GRIFFIN are confusing because they represent thread interactions using treeviews. That is because of our design policy. We designed the three tools to have similar

look-and-feels to minimize the effect of using different UIs. Thus, we used a list view for TRACER and tree views for UNICORN and GRIFFIN. In addition, we used the same color scheme for all tools for consistency. However, that policy confused participants. They discussed better visualization supports of the tools with several interesting insights with the following comments:

> *I feel that if there could be a report of possible bugs (like doc) using descriptive languages will be quite useful. Or it could be also useful if a picture of possible bugs, something like SVN's branch tree, could be also quite useful and make the programmer immediately knows what branches might be wrong. Different colours could be used for differentiating which bugs might be more likely.*

> *Although the tool showed read and write patterns as 'R' and 'W' in the square, but it was not visually well noticeable, thus due to the limited time, I didn't have much chance to pay attention to my analysis. It would be great if that aspect can also be visually more highlighted to better guide users.*

In summary, our tools need better visualization support to improve understanding of concurrency bugs.

6.7.2.2 The tools need to support interactive debugging features

As most participants frequently use breakpoint-based interactive debuggers, they missed some interactive debugging features from our tools. First, many participants complained that it would be much better if the tools show variable's values. However, it is currently difficult to show in UNICORN and GRIFFIN because the techniques are not designed to record complete value information. Also, because values may change in different executions, showing the specific value may be also a problem.

Second, some participants complained that the tool does not provide any connection between the program output and the tool results. That is only possible when the debugger has a complete control over the program execution, but our tool was not designed in that way. Although these suggestions are not directly addressable, the future tool developers may consider these aspects.

6.7.2.3 The tools need to provide fix advice

The survey results showed no correlation between understanding and fix scores. To support complete debugging process, from understanding to fixing concurrency bugs, a future technique may support information on fix advice in addition to information on understanding. For example, one future research direction is to suggest fix patterns for different types of concurrency bugs.

6.8 Summary

A previous study showed that debuggers should provide contextual information to better explain bugs [71]. In previous Chapter 5, we developed Griffin to provide contextual information, but is it really helping developers for understanding concurrency bugs?

To investigate it, this chapter presented an empirical user study that evaluates the effectiveness of existing debugging tools for concurrency bugs for 32 developers. The results showed that Griffin assists better in understanding concurrency bugs than other tools, especially for harder tasks. The study also suggested the following research directions: (1) Debuggers for concurrency bugs need to improve the visualization for user interface, (2) Debuggers for concurrency bugs need to support interactive debugging features, and (3) Debuggers for concurrency bugs need to provide fix advice.

CHAPTER VII

CONCLUSION AND FUTURE DIRECTIONS

The main technical finding of this dissertation is that there exist effective and efficient techniques to isolate specific program fragments that represent the root-cause of a concurrency fault. Our work to develop such techniques culminates in the Griffin technique of Chapter 5. In Chapter 6, we designed a user study which at least partially validated the effectiveness of the techniques. In particular, Griffin's combination of interleaved memory reference and call stack analysis become more effective than other techniques as the concurrency bug itself becomes more complex.

However, this dissertation also raises a number of questions, which we strongly believe can form the basis of future work, as outlined below.

Investigate effects on test inputs The empirical studies in Chapters 3 and 4 showed that Falcon and Unicorn are effective in ranking suspicious memory accesses with rank 1 or 2, for our subjects, but our evaluation relied on a few test inputs and multiple program executions with random delays to make diverse interleavings across multiple executions. For sequential programs, recent work addressed this problem by measuring the effect of test suite on fault localization and then by generating test inputs to improve fault-localization results [36, 78]. Like the approaches, future research can measure the effect of test inputs and interleavings and develop an improved technique to get better fault-localization results.

Extend bug-diagnosis coverage Another area for future work concerns the faulty patterns that our fault-localization techniques detect. Our techniques detect order and atomicity violations, which consist of around 70% of all industrial level concurrency bugs [50]. However, there are additional types of concurrency bugs, such

as deadlock, which can be detected with dynamic analysis [9, 80]. Future work can develop a method to integrate these techniques with our dynamic pattern-based detection techniques to extend the bug-diagnose coverage.

Reduce runtime overhead Current bug-diagnosis approaches, including our techniques, show significant slowdowns over uninstrumented programs. Because these techniques monitor memory accesses between different accesses, the overhead increases as more instrumented memory addresses are accessed. Thus, developing a scheme that efficiently monitors memory accesses during program execution is a future direction. Techniques such as sampling [34], annotation [53], and static analysis [49] could ameliorate the overheads.

Support interactive debugging features The results of empirical user studies in Chapter 6 resulted in many research directions. One noticeable research direction is to support interactive debugging features. Since developers frequently use interactive debuggers for debugging sequential programs, they want to use those features for debugging concurrent programs. The features include watching changes of values for a specific variable, updating values as the developer edits the code, and showing an enhanced view of concurrency bugs within an IDE. Because our techniques are designed to improve understanding the root causes, but not to support interactive features, the fundamental design needs to be changed to support these features. For example, complete (thus heavy) execution information can be collected and saved efficiently to enable the debugger to show the values.

Provide fix advice Another interesting observation from the empirical user studies in Chapter 6 is that our techniques do not suggest any clue on fixing concurrency bugs. However, a future tool can be extended to automatically fix the bug or suggest a fix candidate to the developer. Automatic fix is a challenging direction because concurrency bugs are the most difficult type of software bugs to debug: almost 40%

of initial patches to concurrency bugs are buggy [93]. In addition, existing automatic fix approaches are mostly focused on adding synchronizations [33,44], whereas adding synchronization is a strategy only used 20% among all fix strategies [50]. Thus, suggesting a possible fix and asking for confirmation from the developer might be another promising approach. Future research may focus on suggesting fix patterns.

APPENDIX A

SUBJECT PROGRAMS

A.1 Bank Account

Description: The program creates several account users, where each user has own account. The program performs three operations (deposit, withdraw, and transfer) for each user concurrently.

Passing output: Each account has $300 at the end.

```
The initial values:
The final values:
A–300.0
B–300.0
```

Failing output: The final amount is not $300.

```
The initial values:
The final values:
A–200.0
B–300.0
```

A.2 Shop

Description: The program has one supplier and multiple customers: The supplier provides items to the shop; The customers buy the items from the shop.

Passing output: The program exits without any exception. Here is sample "output.txt".

```
<shop, customer 2: 10 customer 1: 10, no bug>
```

Failing output: 1. The program prints "java.lang.ArrayIndexOutOfBoundsException: -1". Here is sample "output.txt".

```
<shop, customer 2: -1 customer 1: 10, denail ( init sleep) + weak reality (lock unlock lock)>
```

A.3 List

Description: It is a part of Java Collection Library of JDK 1.4. In this program, it creates several List objects and performs multiple operations (add, addAll, remove, containsAll) concurrently.

Passing output: The program prints nothing

Failing output: The program crashes with the following exception.

```
java.lang.RuntimeException: Bug found. l1 has a null element. : [2, 3, 4, 5, 6, null]
```

APPENDIX B

SURVEY

B.1 Background

1. Please enter your UUID.
2. What is your programming experience in number of years?
3. What is your Java experience in number of years?
4. What is your experience on concurrency?
   (a) No experience
   (b) Beginner
   (c) Used in a class project
   (d) Used in projects
   (e) Expert
5. What is your current job?
   (a) Undergraduate student
   (b) Master student
   (c) PhD student
   (d) Professional programmer

B.2 Task 1: Bank Account

1. Please enter your UUID.
2. Which tool did you use to debug this subject?
   (a) Tracer
   (b) Unicorn

(c) Griffin

3. What is the type of the bug?

   (a) Order violation

   (b) Atomicity violation

   (c) Other concurrency bug

   (d) I don't know

4. Root cause (variable level): Select one or more variables that are responsible for the root cause of the bug.

   (a) Account.amount

   (b) Account.name

   (c) AccountUser.accNum

   (d) AccountUser.maxaccounts

5. Root cause (method level): Select one or more methods that are responsible for the root cause of the bug.

   (a) Account.deposit

   (b) Account.withdraw

   (c) Account.transfer

   (d) Account.print

   (e) AccountUser.errorCheck

   (f) AccountUser.printAllAccounts

6. Root cause (interleaving): Please describe the bug scenario (access orders between threads).

7. What is your fix strategy of the bug?

   (a) Add synchronization

   (b) Add condition check

(c) Switch order of statements

(d) Other strategies

8. Please describe your possible fix.
9. Is the tool useful? (1,low to 5,high)
10. Are you confident with the result? (1,low to 5,high)
11. Please give us any feedback on this task.

B.3 Task 2: Shop

1. Please enter your UUID.
2. Which tool did you use to debug this subject?

   (a) Tracer

   (b) Unicorn

   (c) Griffin

3. What is the type of the bug?

   (a) Order violation

   (b) Atomicity violation

   (c) Other concurrency bug

   (d) I don't know

4. Root cause (variable level): Select one or more variables that are responsible for the root cause of the bug.

   (a) Shop.items

   (b) Shop.storage

   (c) Supplier.supply_address

   (d) Customer.shopping_list

   (e) Signal.i

5. Root cause (method level): Select one or more methods that are responsible for the root cause of the bug.
   (a) Shop.isEmpty
   (b) Shop.getItem
   (c) Shop.putItem
   (d) Signal.set
   (e) Signal.get
   (f) Customer.buy
   (g) Supply.supply
6. Root cause (interleaving): Please describe the bug scenario (access orders between threads).
7. What is your fix strategy of the bug?
   (a) Add synchronization
   (b) Add condition check
   (c) Switch order of statements
   (d) Other strategies
8. Please describe your possible fix.
9. Is the tool useful? (1,low to 5,high)
10. Are you confident with the result? (1,low to 5,high)
11. Please give us any feedback on this task.

B.4 Task 3: List

1. Please enter your UUID.
2. Which tool did you use to debug this subject?
   (a) Tracer
   (b) Unicorn

(c) Griffin

3. What is the type of the bug?

   (a) Order violation

   (b) Atomicity violation

   (c) Other concurrency bug

   (d) I don't know

4. Root cause (variable level): Select one or more variables that are responsible for the root cause of the bug.

   (a) ArrayList.size

   (b) ArrayList.mutex

   (c) ArrayList.elementData

   (d) ArrayList.modCount

   (e) ArrayList.cursor

5. Root cause (method level): Select one or more methods that are responsible for the root cause of the bug.

   (a) ArrayList.toArray

   (b) ArrayList.addAll

   (c) ArrayList.<init>

   (d) ArrayList.remove

   (e) ArrayList.ensureCapacity

   (f) ArrayList.size

   (g) Main.run

6. Root cause (interleaving): Please describe the bug scenario (access orders between threads).

7. What is your fix strategy of the bug?

(a) Add synchronization

(b) Add condition check

(c) Switch order of statements

(d) Other strategies

8. Please describe your possible fix.
9. Is the tool useful? (1,low to 5,high)
10. Are you confident with the result? (1,low to 5,high)
11. Please give us any feedback on this task.

B.5 Feedback

1. Please enter your UUID.
2. Please rank the three tools by usefulness and discuss why.
3. Please give us any feedback on the study.

REFERENCES

[1] "Chess: Systematic concurrency testing." http://chesstool.codeplex.com/.

[2] "Factorial experiment - wikipedia." http://en.wikipedia.org/wiki/Factorial_experiment.

[3] "Nasdaq's facebook glitch came from race conditions." http://www.pcworld.com/businesscenter/article/255911/nasdaqs_facebook_glitch_came_from_race_conditions.html.

[4] "Ranking - wikipedia." http://en.wikipedia.org/wiki/Ranking.

[5] "Within-subject design - wikipedia." http://en.wikipedia.org/wiki/Within-subject_design.

[6] Abreu, R., Zoeteweij, P., and van Gemund, A. J. C., "On the accuracy of spectrum-based fault localization," in *Proceedings of the The Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)*, pp. 89–98, Sept. 2007.

[7] Artho, C., Havelund, K., and Honiden, S., "Visualization of concurrent program executions," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 2, pp. 541–546, 2007.

[8] Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J. D., Lee, E. A., Morgan, N., Necula, G., Patterson, D. A., Sen, K., Wawrzynek, J., Wessel, D., and Yelick, K. A., "The parallel computing laboratory at u.c. berkeley: A research agenda based on the berkeley view," tech. rep., EECS Department, University of California, Berkeley, 2008.

[9] Bensalem, S. and Havelund, K., "Dynamic deadlock analysis of multi-threaded programs," in *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, pp. 208–223, 2006.

[10] Blackburn, S. M., Garner, R., Hoffmann, C., Khang, A. M., McKinley, K. S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S. Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J. E. B., Moss, B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., and Wiedermann, B., "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proceedings of 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pp. 169–190, ACM, Oct. 2006.

[11] Chen, Q., Wang, L., Yang, Z., and Stoller, S. D., "HAVE: detecting atomicity violations via integrated dynamic and static analysis," in *ETAPS*, pp. 425–439, 2009.

[12] Cheng, H., Lo, D., Zhou, Y., Wang, X., and Yan, X., "Identifying bug signatures using discriminative graph mining," in *Proc. of Intl Symp. on Softw. Testing and Analy.*, pp. 141–152, July 2009.

[13] Dang, Y., Wu, R., Zhang, H., Zhang, D., and Nobel, P., "ReBucket: A method for clustering duplicate crash reports based on call stack similarity," in *ICSE*, pp. 1084–1093, June 2012.

[14] DeSouza, J., Kuhn, B., de Supinski, B. R., Samofalov, V., Zheltov, S., and Bratanov, S., "Automated, scalable debugging of MPI programs with the Intel® Message Checker," in *Proc. Int'l. Wkshp. Software Eng. for HPC System Applications*, pp. 78–82, May 2005.

[15] Eccles, R., Nonneck, B., and Stacey, D., "Exploring parallel programming knowledge in the novice," in *High Performance Computing Systems and Applications, 2005. HPCS 2005. 19th International Symposium on*, pp. 97–102, 2005.

[16] Eccles, R. and Stacey, D., "Understanding the parallel programmer," in *High-Performance Computing in an Advanced Collaborative Environment, 2006. HPCS 2006. 20th International Symposium on*, pp. 12–12, 2006.

[17] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., and Ur, S., "Multithreaded java program test generation," in *Java Grande-ISCOPE*, 2001.

[18] Eytani, Y., Havelund, K., Stoller, S. D., and Ur, S., "Towards a framework and a benchmark for testing tools for multi-threaded programs," *Concurr. Comput. : Pract. Exper.*, vol. 19, no. 3, pp. 267–279, 2007.

[19] Flanagan, C. and Freund, S. N., "Type-based race detection for java," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 219–232, June 2000.

[20] Flanagan, C. and Freund, S. N., "Atomizer: A dynamic atomicity checker for multithreaded programs," in *Proc. of Symp. on Principles of Programming Languages*, pp. 256–267, January 2004.

[21] Flanagan, C., Freund, S. N., and Yi, J., "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 293–303, June 2008.

[22] Flanagan, C. and Qadeer, S., "A type and effect system for atomicity," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 338–349, June 2003.

[23] Godefroid, P. and Nagappan, N., "Concurrency at Microsoft: An exploratory survey," in *Wkshop. on (EC)2*, July 2008.

[24] Halpert, R. L., "Static lock allocation," Master's thesis, McGill University, 2008.

[25] Hammer, C., Dolby, J., Vaziri, M., and Tip, F., "Dynamic detection of atomic-set-serializability violations," in *Proc. of Int'l Conf. on Softw. Eng*, pp. 231–240, May 2008.

[26] Havelund, K. and Pressburger, T., "Model checking java programs using java pathfinder," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 366–381, 2000.

[27] Hochstein, L., Carver, J., Shull, F., Asgari, S., Basili, V., Hollingsworth, J. K., and Zelkowitz, M. V., "Parallel programmer productivity: A case study of novice parallel programmers," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, (Washington, DC, USA), pp. 35–, IEEE Computer Society, 2005.

[28] Hsu, H.-Y., Jones, J. A., and Orso, A., "Rapid: Identifying bug signatures to support debugging activities," in *Proc. of Intl Conf. on Automated Softw. Eng.*, pp. 439–442, Sept. 2008.

[29] Huang, J. and Zhang, C., "Persuasive prediction of concurrency access anomalies," in *Proc. of Intl Symp. on Softw. Testing and Analy.*, pp. 144–154, July 2011.

[30] Hussain, I., Csallner, C., Grechanik, M., Fu, C., Xie, Q., Park, S., Taneja, K., and Hossain, B. M. M., "Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator," in *Proceedings of the Workshop on Dynamic Analysis*, July 2012.

[31] Jalbert, N., Pereira, C., Pokam, G., and Sen, K., "RADBench: A concurrency bug benchmark suite," in *Proc. of HotPar*, May 2011.

[32] Jalbert, N. and Sen, K., "A trace simplification technique for effective debugging of concurrent programs," in *Proceedings of Symposium on Foundations of Software Engineering*, pp. 57–66, Nov. 2010.

[33] Jin, G., Song, L., Zhang, W., Lu, S., and Liblit, B., "Automated atomicity-violation fixing," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 389–400, June 2011.

[34] Jin, G., Thakur, A., Liblit, B., and Lu, S., "Instrumentation and sampling strategies for cooperative concurrency bug isolation," in *Proc. of Intl Conf. on Object-Orientted Prog., Systt., Lang., and Apps.*, pp. 241–255, Oct. 2010.

[35] Jin, G., Zhang, W., Deng, D., Liblit, B., and Lu, S., "Automated concurrency-bug fixing," in *OSDI*, pp. 221–236, Oct. 2012.

[36] Jin, W. and Orso, A., "F3: fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pp. 213–223, 2013.

[37] Jones, J. A., Bowring, J. F., and Harrold, M. J., "Debugging in parallel," in *Proc. of Intl Symp. on Softw. Testing and Analy.*, pp. 16–26, July 2007.

[38] Jones, J. A. and Harrold, M. J., "Empirical evaluation of the tarantula automatic. Fault-Localization technique," in *Proc. of Intl Conf. on Automated Softw. Eng.*, pp. 273–282, Nov. 2005.

[39] Jones, J. A., Harrold, M. J., and Stasko, J., "Visualization of test information to assist fault localization," in *Proc. of Intl Conf. on Softw. Eng*, pp. 467–477, May 2002.

[40] Ko, A. J. and Myers, B. A., "Debugging reinvented: Asking and answering why and why not questions about program behavior," in *Proceedings of the 30th International Conference on Software Engineering*, pp. 301–310, 2008.

[41] Ko, A. J. and Myers, B. A., "Finding causes of program output with the java whyline," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '09, pp. 1569–1578, 2009.

[42] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable statistical bug isolation," in *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 15–26, 2005.

[43] Liu, C., Yan, X., Fei, L., Han, J., and Midkiff, S. P., "Sober: statistical model-based bug localization," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 286–295, Sept. 2005.

[44] Liu, P. and Zhang, C., "Axis: Automatically fixing atomicity violations through solving control constraints," in *Proc. of Intl Conf. on Softw. Eng*, pp. 299–309, June 2012.

[45] Lonnberg, J. and Berglund, A., "Students' understandings of concurrent programming," in *Seventh Baltic Sea Conference on Computing Education Research (Koli Calling 2007)* (Lister, R. and Simon, eds.), vol. 88 of *CRPIT*, (Koli National Park, Finland), pp. 77–86, ACS, 2007.

[46] Lonnberg, J., Berglund, A., and Malmi, L., "How students develop concurrent programs," in *Eleventh Australasian Computing Education Conference (ACE 2009)* (Hamilton, M. and Clear, T., eds.), vol. 95 of *CRPIT*, (Wellington, New Zealand), pp. 129–138, ACS, 2009.

[47] Lönnberg, J., Malmi, L., and Ben-Ari, M., "Evaluating a visualisation of the execution of a concurrent program," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, Koli Calling '11, (New York, NY, USA), pp. 39–48, ACM, 2011.

[48] Lönnberg, J., Malmi, L., and Berglund, A., "Helping students debug concurrent programs," in *Proceedings of the 8th International Conference on Computing Education Research*, Koli '08, (New York, NY, USA), pp. 76–79, ACM, 2008.

[49] Lu, S., Park, S., Hu, C., Ma, X., Jiang, W., Li, Z., Popa, R. A., and Zhou, Y., "MUVI: automatically inferring multi-variable access correlations and. detecting related semantic and concurrency bugs," in *SOSP*, pp. 103–116, 2007.

[50] Lu, S., Park, S., Seo, E., and Zhou, Y., "Learning from mistakes—A comprehensive study on real world concurrency bug characteristics," in *Proc. of Int'l Conf. on Arch. Supp. for Prog. Lang. and Oper. Syst.*, pp. 329–339, Mar. 2008.

[51] Lu, S., Tucek, J., Qin, F., and Zhou, Y., "AVIO: Detecting atomicity violations via access interleaving invariants," in *Proc. of Intl Conf. on Arch. Supp. for Prog. Lang. and Oper. Syst.*, pp. 37–48, Oct. 2006.

[52] Lucia, B. and Ceze, L., "Finding concurrency bugs with context-aware communication graphs," in *Intl Symp. on Microarchitecture*, pp. 553–563, Dec. 2009.

[53] Lucia, B., Ceze, L., and Strauss, K., "ColorSafe: Architectural support for debugging and dynamically avoiding multi-variable atomicity violation," in *Proc. of Intl Symp. on Comp. Arch.*, pp. 222–233, June 2010.

[54] Lucia, B., P. Wood, B., and Ceze, L., "Isolating and understanding concurrency errors using reconstructed execution fragments," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 378–388, June 2011.

[55] Maheswara, G., Bradbury, J. S., and Collins, C., "Tie: an interactive visualization of thread interleavings.," in *SOFTVIS*, pp. 215–216, 2010.

[56] McDowell, C. E. and Helmbold, D. P., "Debugging concurrent programs," *ACM Comp. Surveys*, vol. 21, no. 4, pp. 593–622, 1989.

[57] Musuvathi, M. and Qadeer, S., "Iterative context bounding for systematic testing of multithreaded programs," *SIGPLAN Not.*, pp. 446–455, June 2007.

[58] Musuvathi, M. and Qadeer, S., "Fair stateless model checking," in *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, (Tucson, AZ, USA), pp. 362–371, ACM, 2008.

[59] Muzahid, A., Otsuki, N., and Torrellas, J., "AtomTracker: a comprehensive approach to atomic region inference and violation detection," in *MICRO*, pp. 287–297, 2010.

[60] Naik, M. and Aiken, A., "Conditional must not aliasing for static race detection," in *Proc. of Symp. on Principles of Programming Languages*, pp. 327–338, January 2007.

[61] Naik, M., Aiken, A., and Whaley, J., "Effective static race detection for java," *SIGPLAN Not.*, pp. 308–319, June 2006.

[62] Nanz, S., West, S., and da Silveira, K. S., "Examining the expert gap in parallel programming," in *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, (Berlin, Heidelberg), pp. 434–445, Springer-Verlag, 2013.

[63] Narayanasamy, S., Wang, Z., Tigani, J., Edwards, A., and Calder, B., "Automatically classifying benign and harmful data racesallusing replay analysis," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 22–31, June 2007.

[64] O'Callahan, R. and Choi, J.-D., "Hybrid dynamic data race detection," in *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 167–178, 2003.

[65] Park, C. and Sen, K., "Randomized active atomicity violation detection in concurrent programs," in *Proc. of Symp. on Foundations of Softw. Eng*, pp. 135–145, Nov. 2008.

[66] Park, S., Harrold, M. J., and Vuduc, R., "Griffin: Grouping suspicious memory-access patterns to improve understanding of concurrency bugs," in *ISSTA*, pp. 134–144, July 2013.

[67] Park, S., Vuduc, R., and Harrold, M. J., "Falcon: Fault localization in concurrent programs," in *Proc. of Intl Conf. on Softw. Eng*, pp. 245–254, May 2010.

[68] Park, S., Vuduc, R., and Harrold, M. J., "A unified approach for localizing non-deadlock concurrency bugs," in *Proc. of Int'l Conf. on Softw. Testing, Verification, and Validation*, pp. 51–60, Apr. 2012.

[69] Park, S., Vuduc, R., and Harrold, M. J., "Unicorn: a unified approach for localizing non-deadlock concurrency bugs," *Software Testing, Verification and Reliability*, 2014.

[70] Park, S., Lu, S., and Zhou, Y., "CTrigger: Exposing atomicity violation bugs from their hiding places," in *Proc. of Intl Conf. on Arch. Supp. for Prog. Lang. and Oper. Syst.*, Mar. 2009.

[71] Parnin, C. and Orso, A., "Are automated debugging techniques actually helping programmers?," in *Proc. of Intl Symp. on Softw. Testing and Analy.*, pp. 199–209, July 2011.

[72] Podgurski, A., Leon, D., Francis, P., Masri, W., Minch, M., Sun, J., and Wang, B., "Automated support for classifying software failure reports," in *Proc. of Intl Conf. on Softw. Eng*, pp. 465–475, May 2003.

[73] Poulsen, K., "Tracking the blackout bug," *SecurityFocus*, February 2004. `http://www.securityfocus.com/news/8412`.

[74] Reiss, S. and Renieris, M., "Demonstration of jive and jove: Java as it happens," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 662–663, 2005.

[75] Reiss, S. P. and Karumuri, S., "Visualizing threads, transactions and tasks," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pp. 9–16, 2010.

[76] Renieres, M. and Reiss, S., "Fault localization with nearest neighbor queries," in *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pp. 30–39, Oct. 2003.

[77] Ronsse, M. and Bosschere, K. D., "RecPlay: A fully integrated practical record/replay system," *Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.

[78] Rössler, J., Fraser, G., Zeller, A., and Orso, A., "Isolating failure causes through test case generation," in *Proc. of Intl Symp. on Softw. Testing and Analy.*, pp. 309–319, July 2012.

[79] Sadowski, C. and Yi, J., "User evaluation of correctness conditions: A case study of cooperability," in *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, (New York, NY, USA), pp. 2:1–2:6, ACM, 2010.

[80] Samak, M. and Ramanathan, M. K., "Trace driven dynamic deadlock detection and reproduction," in *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 29–42, 2014.

[81] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T., "Eraser: A dynamic data race detector for multithreaded programs," *Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[82] Sen, K., "Race directed random testing of concurrent programs," in *Proc. of ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pp. 11–21, June 2008.

[83] Shi, Y., Park, S., Yin, Z., Lu, S., Zhou, Y., Chen, W., and Zheng, W., "Do I use the wrong definition? DefUse: Definition-use invariants for detecting concurrency and sequential bugs," in *Proc. of Int'l Conf. on Object-Orientted Prog., Systt., Lang., and Apps.*, pp. 160–174, Oct. 2010.

[84] Sorrentino, F., Farzan, A., and Madhusudan, P., "PENELOPE: weaving threads to expose atomicity violations," in *FSE*, pp. 37–46, 2010.

[85] Steinbach, M., Karypis, G., and Kumar, V., "A comparison of document clustering techniques," in *Workshop on Text Mining*, p. 525, Aug. 2000.

[86] Süss, M. and Leopold, C., "Common mistakes in OpenMP and how to avoid them," in *OpenMP Shared Memory Parallel Programming*, vol. LNCS 4315, pp. 312–323, 2008.

[87] Szafron, D. and Schaeffer, J., "Experimentally assessing the usability of parallel programming systems," in *Programming Environments for Massively Parallel Distributed Systems* (Decker, K. and Rehmann, R., eds.), Monte Verit, pp. 195–201, Birkhuser Basel, 1994.

[88] Vaziri, M., Tip, F., and Dolby, J., "Associating synchronization constraints with data in an Object-Oriented language," in *Proc. of Symp. on Principles of Programming Languages*, pp. 334–345, 2006.

[89] Vuduc, R., Schulz, M., Quinlan, D., de Supinski, B., and Sbjrnsen, A., "Improving distributed memory applications testing by message perturbation," in *Proceedings of the workshop on Parallel and distributed systems: testing and debugging (PADTAD)*, pp. 27–36, July 2006.

[90] Wang, L. and Stoller, S., "Runtime analysis of atomicity for multithreaded programs," *Software Engineering, IEEE Transactions on*, vol. 32, pp. 93–110, Feb. 2006.

[91] Weiser, M., "Program slicing," in *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pp. 439–449, 1981.

[92] Yi, J., Sadowski, C., and Flanagan, C., "Sidetrack: Generalizing dynamic atomicity analysis," in *PADTAD*, 2009.

[93] Yin, Z., Yuan, D., Zhou, Y., Pasupathy, S., and Bairavasundaram, L., "How do fixes become bugs?—A comprehensive characteristic study on incorrect fixes in commercial and open source operating systems," in *Proc. of Symp. on Foundations of Softw. Eng*, pp. 26–36, Sept. 2011.

[94] Yu, J. and Narayanasamy, S., "A case for an interleaving constrained Shared-Memory Multi-Processor," in *Proceedings of International Symposium on Computer Architecture*, pp. 325–336, June 2009.

[95] Yu, J. and Narayanasamy, S., "Tolerating concurrency bugs using transactions as lifeguards," in *MICRO*, pp. 263–274, 2010.

[96] Zeller, A., *Why Programs Fail: A Guide to Systematic Debugging*. 2009.

[97] Zhang, W., Lim, J., Olichandran, R., Scherpelz, J., Jin, G., Lu, S., and Reps, T., "ConSeq: detecting concurrency bugs through sequential errors," in *Proc. of Intl Conf. on Arch. Supp. for Prog. Lang. and Oper. Syst.*, pp. 251–264, Mar. 2011.

[98] Zheng, A. X., Jordan, M. I., Liblit, B., Naik, M., and Aiken, A., "Statistical debugging: Simultaneous identification of multiple bugs," in *Proc. of Intl Conf. on Machine Learning*, pp. 1105–1112, June 2006.